Imperial College London
Department of Computing

Automated Optimization of Numerical Methods for Partial Differential Equations

Fabio Luporini

May 2016

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

Contents

1 Introduction **1**

1.1 Thesis Statement 1

1.2 Overview 1

1.3 Contributions 2

1.4 Dissemination 2

1.5 Thesis Outline 3

2 Background **5**

2.1 The Finite Element Method 5

2.1.1 Variational Formulation 5

2.1.2 Mapping from the Reference Element 5

2.1.3 Assembly 5

2.1.4 Linear Solvers 10

2.1.5 Impact of Assembly on Execution Time 10

2.2 Compilers and Libraries for Code Optimization 11

2.3 State-of-the-art Hardware Architectures 11

3 Automated Tiling for Irregular Computations **13**

4 On Optimality of Finite Element Integration **15**

4.1 Introduction 15

4.2 Rewrite Operators 15

5 **Cross-loop Optimization of Arithmetic Intensity for Finite Element Integration** **17**
5.1 Recapitulation and Objectives 17
5.2 Low-level Optimization 20
5.2.1 Padding and Data Alignment 20
5.2.2 Expression Splitting 21
5.2.3 Model-driven Vector-register Tiling 23
5.2.4 Vector-expansion of Invariant Sub-expressions 25
5.3 Experiments 25
5.3.1 Experimental Setup 25
5.3.2 Rationale of the Study 25
5.3.3 Impact of Transformations 27
5.4 Experience with General-purpose Compiler Optimizations . 31
5.4.1 Loop Interchange 31
5.4.2 Loop Unroll 32
5.5 Related Work 32
5.6 Generality of the Approach and Applicability to Other Domains 34
5.7 Conclusion 38

6 **COFFEE: a Compiler for Fast Expression Evaluation** **41**
6.1 Overview 41
6.2 Outline and Integration with Firedrake 41
6.3 Modifying the FEniCS Form Compiler 43
6.3.1 Generation of Abstract Syntax Trees 43
6.4 Implementation 44
6.4.1 Code Hoisting 46
6.4.2 Benefits of Tracking Data Dependencies 47

7 **Conclusions** **49**

List of Tables

2.1 Type and variable names used in the various listings to identify local assembly objects. 8

List of Figures

2.1 Structure of a local assembly kernel 7

5.1 Outer-product vectorization by permuting values in a vector register. 24
5.2 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. 24
5.3 Performance improvement due to generalized loop-invariant code motion (*licm*), data alignment and padding (*ap*), outer-product vectorization (*op-vect*), and expression splitting (*split*) over the original non-optimized code. In each plot, the horizontal axis reports speed ups, whereas the polynomial order q of the method varies along the vertical axis. 28

6.1 High-level view of Firedrake. COFFEE is at the core, receiving ASTs from a modified version of the FEniCS Form Compiler and producing optimized C code kernels. 42
6.2 AST representation of a C assignment in COFFEE. 44
6.3 Structure of COFFEE. 45

Chapter 1

Introduction

1.1 Thesis Statement

1.2 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.

1.3 Contributions

1.4 Dissemination

The research exposed in this thesis has been disseminated in the scientific community through various channels:

- **Papers.** The following is the list of publications derived from the research activity (chronological order):
  1. Strout, M.M.; Luporini, F.; Krieger, C.D.; Bertolli, C.; Bercea, G.-T.; Olschanowsky, C.; Ramanujam, J.; Kelly, P.H.J., ”Generalizing Run-Time Tiling with the Loop Chain Abstraction,” Parallel and Distributed Processing Symposium, 2014 IEEE 28th International , vol., no., pp.1136,1145, 19-23 May 2014
  2. Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. ”Cross-loop optimization of arithmetic intensity for finite element local assembly”. 2014. Submitted for publication.
  3. Fabio Luporini, David A. Ham, Paul H. J. Kelly. ”Optimizing Automated Finite Element Integration through Expression Rewriting and Code Specialization”. 2014. To be written.
- **Talks.** Talks have been delivered at the following conferences/workshops:

    1. ”Generalised Sparse Tiling for Unstructured Mesh Computations in the OP2 Framework”. Compilers for Parallel Computing, July 2013.
    2. ”COFFEE: an Optimizing Compiler for Fintie Element Local Assembly”. FEniCS Workshop, July 2014.

- **Software.** The following software is released under open source licenses.
    1. COFFEE (COmpiler For Finit Element local assEmbly), the compiler described in Chapter 6.

, and the design of this software and results have been disseminated in the scientific community through publications.

1.5 Thesis Outline

Chapter 2

Background

2.1 The Finite Element Method

...

2.1.1 Variational Formulation

...

2.1.2 Mapping from the Reference Element

...

2.1.3 Assembly

...

From Math to Loop Nests

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 **??**), whose sizes depend on the order of the method. This operation is applied to all cells in the discretized domain (mesh).

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

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

  // 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 W[6] = 0.5;

  // Basis functions
  static const double B[6][3] = {{...}} ;
  static const double C[6][3] = {{...}} ;
  static const double D[6][3] = {{...}} ;

  for (int i = 0; i < 6; ++i) {
    double f0 = 0.0;
    for (int r = 0; r < 3; ++r) {
      f0 += (w[r] * C[i][r]);
    }
    for (int j = 0; j < 3; ++j) {
      for (int k = 0; k < 3; ++k) {
        A[j][k] += ((((K[1]*B[i][k])+(K[3]*D[i][k])) *
                     ((K[1]*B[i][j])+(K[3]*D[i][j]))) +
                    (((K[0]*B[i][k])+(K[2]*D[i][k])) *
                     ((K[0]*B[i][j])+(K[2]*D[i][j]))))*det*W[i]*f0);
      }
    }
  }
}
```

We consider again the weighted Laplace example of the previous section. A C-code implementation of Equation **??** is illustrated in Listing 1. The values at the various quadrature points of basis functions (ϕ) derivatives are tabulated in the A and B 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

LISTING 2: UFL specification of the weighted Laplace equation for polynomial order $q = 2$ 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 2.1: Structure of a local assembly kernel

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 to obtain with minimum effort a semantically correct implementation of local assembly. The goal of this research is maximizing the efficiency, in terms of run-time performance, of generic local assembly kernels, 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., 2014]. Listing 2 shows a possible UFL implementation for the weighted Laplace form. Note the resemblance of *a = weight*...* with Equation **??**. A form compiler translates UFL code into the C code shown in Listing 1. We will describe these aspects carefully in Section 6.2; for the moment, this level of detail sufficies to open a discussion on the optimization of local assembly kernels arising from different partial differential equations.

The structure of a local assembly kernel can be generalized as in Figure 2.1. 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.

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
Coefficients at quadrature points	double	f0, f1, ...
Numerical integration weights	double[]	W
Basis functions (and derivatives)	double[][]	A, B, C, ...
Element matrix	double[][]	M

Table 2.1: Type and variable names used in the various listings to identify local assembly objects.

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

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 $q = 1$. 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)

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

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

  static const double W[3] = {...}
  static const double A[3][3] = {{...}}
  static const double B[3][3] = {{...}}

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

to act as scaling factors of arrays. Trip counts grow proportionally to the order of the method and arrays may be block-sparse.

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

LISTING 4: Local assembly implementation for a Burgers problem on a 3D mesh using polynomial order $q = 1$ 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 A[5][12] = {{...}}
 static const double B[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]*C[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*C[i][k])+(K3*D[i][k])+(K6*A[i][k]))*Y2[i][j]))*f11)+
           +(((K2*E[i][k])+...+(K8*B[i][k]))*((K2*E[i][j])+...+(K8*B[i][j])))+
           + <roughly a hundred sum/muls go here>)..)*
           *det*W[i];
 }
}
```

three loop dimensions are always unit stride; 2) the `j` and `k` loops are interchangeable, whereas interchanges involving the i loop require pre-computation 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.

2.1.4 Linear Solvers

..

2.1.5 Impact of Assembly on Execution Time

...

2.2 Compilers and Libraries for Code Optimization

...

2.3 State-of-the-art Hardware Architectures

...

Chapter 3

Automated Tiling for Irregular Computations

...

Chapter 4

On Optimality of Finite Element Integration

4.1 Introduction

...

4.2 Rewrite Operators

...

...

Chapter 5

Cross-loop Optimization of Arithmetic Intensity for Finite Element Integration

5.1 Recapitulation and Objectives

In Chapter 4, we have developed a method to minimize the operation count of finite element operators, or "assembly kernels". This chapter focuses on the same class of kernels, but tackles an orthogonal issue: low level optimization of generated code. We will abstract from the mathematical structure inherent in the expressions and concentrate on the aspects impacting the computational efficiency.

We know that 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 register locality and 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 5.6, however, we provide some intuitions about this research direction.

Achieving high-performance on CPUs is non-trivial. The complexity of the mathematical expressions, which we know to be often characterized by a large number of operations on constants and small vectors, 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. The complexity of the memory access pattern also depends on the kernel, specifically on the function spaces employed by the method, ranging from unit-stride (e.g., `A[i]`, `A[i+1]`, `A[i+2]`, `A[i+3]`, ...) to random-stride (e.g., `A[i]`, `A[i+1]`, `A[i+2]`, `A[i+N]`, `A[i+N+1]`, ...). We will show that traditional vendor compilers, such as *GNU's* and *Intel's*, fail at maximizing the efficiency of the generated code because of such a particular structure. 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 since they mostly focus on cache locality.

Like in Chapter 4, we focus on optimizing the performance of assembly kernels produced through automated code generation, so we seek transformations that are generally applicable and effective. In particular, we will study the following transformations:

Padding and data alignment SIMD vectorization is more effective when the CPU registers are packed (unpacked) by means of aligned load (store) instructions. Data alignment is achieved through array padding, a conceptually simple yet powerful transformation that can result in dramatic reductions in execution time. We will see that the complexity of the transformation increases if non unite-stride memory accesses are present.

Vector-register tiling Blocking at the level of vector registers aims to improve data locality. This transformation exploits the peculiar memory access pattern inherent in finite element operators (i.e., inner products involving test and trial functions).

Expression splitting Complex expressions are often characterized by high register pressure (i.e., the lack of available registers inducing the compiler to "spill" data from registers to cache). This happens, for example, when the number of arrays (e.g., basis functions, temporaries introduced by generalized code motion, temporaries produced by pre-evaluation) and constants is large compared to the number of available registers (typically 16 on state-of-the-art CPUs, 32 on future generations). This transformation exploits the associativity of addition to distribute, or "split", an expression into multiple sub-expressions; each sub-expression is then computed in a separate loop nest.

Vector-expansion of invariant sub-expressions In Chapter 4 we have explained that some sub-expressions can be hoisted at the level of the integration loop. This transformation "trades" space for exposing a parallel dimension (the integration loop), thus promoting SIMD vectorization at the level of an outer loop.

To summarize, the contributions of this chapter are as follows:

- A number of low level transformations for optimizing the performance of assembly kernels.
- Extensive experimentation using a set of real-world forms commonly arising in finite element methods.
- A discussion concerning the generality of the transformations and their applicability to different domains.

5.2 Low-level Optimization

5.2.1 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 (but the element matrix, for reasons discussed shortly) are padded by rounding the innermost dimension 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 32-byte long vector registers and 8-byte double-precision floats). In this case, a padded version of the array will have size 3×4. Secondly, their base address is enforced to multiples of `VL` by means of special attributes. 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.

In our computational model, the element matrix is one of the kernel's input parameters, so it needs special handling when padding (the signature of the kernel must not be changed, otherwise the abstraction would be broken). We create a "shadow" copy of the element matrix, padded, aligned, and initialized to 0. The shadow element matrix is used in place of the original element matrix. Right before returning to the caller, a loop nest copies, discarding the padded region, the shadow matrix back into the input buffer.

Array padding 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.

Figure 5 illustrates the effect of padding and data alignment on top of generalized code motion applied to the weighted Laplace operator presented in Figure 1.

LISTING 5: The assembly kernel for the weighted Laplace operator in Figure 1 after application of padding and data alignment (on top of generalized code motion). An AVX architecture, which implies `VL` = 4, is assumed.

```
void weighted_laplace(double A[3][3], double **coords, double w[3]) {
  #define ALIGN __attribute__((aligned(32)))
  // K, det = Compute Jacobian (coords)

  // Quadrature weights
  static const double W[6] ALIGN = 0.5;

  // Basis functions
  static const double B[6][4] ALIGN = {{...}} ;
  static const double C[6][3] ALIGN = {{...}} ;
  static const double D[6][4] ALIGN = {{...}} ;

  // Padded buffer
  double _A[3][4] ALIGN = {{0.0}};

  for (int i = 0; i<6; i++) {
    double f0 = 0.0;
    for (int r = 0; r < 3; ++r) {
      f0 += (w[r] * C[i][r]);
    }
    double T_0[4] ALIGN;
    double T_1[4] ALIGN;
    for (int k = 0; k<4; r++) {
      T_0[r] = ((K[1]*B[i][k])+(K[3]*D[i][k]));
      T_1[r] = ((K[0]*B[i][k])+(K[2]*D[i][k]));
    }
    for (int j = 0; j<3; j++) {
      #pragma vector aligned
      for (int k = 0; k<4; k++) {
        _A[j][k] += (T_0[k]*T_0[j] + T_1[k]*T_1[j])*det*W[i]*f0);
      }
    }
  }
  for (int j = 0; j<3; j++) {
    for (int k = 0; k<3; k++) {
      A[j][k] = _A[j][k];
    }
  }
}
```

5.2.2 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 problem, is given in Listing **??**. 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.

LISTING 6: Local assembly code generated by Firedrake for the Helmholtz problem in which *split* has been applied on top of the optimizations shown in Listing **??**. In this example, the split factor is 2.

```
void helmholtz(double A[3][4], double **coords) {
   // Same code as in Listing ?? up to the j loop
   for (int j = 0; j<3; j++)
      #pragma vector aligned
      for (int k = 0; k<4; k++)
         A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j])*det*W3[i];
   for (int j = 0; j<3; j++)
      #pragma vector aligned
      for (int k = 0; k<4; k++)
         A[j][k] += LI_1[k]*LI_1[j]*det*W3[i];
}
```

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 and the way expression rewriting is performed, 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, again, depends on expression rewriting. Nevertheless, as shown in Section **??**, the performance gain from improved register reuse along inner dimensions can be important, especially if the selection of the split factor uses heuristics to minimize the aforementioned issues.

5.2.3 Model-driven Vector-register Tiling

LISTING 7: Local assembly code generated by Firedrake for the Helmholtz problem after application of *op-vect* on top of the optimizations shown in Listing **??**. In this example, the unroll-and-jam factor is 1.

```
void helmholtz(double A[4][4], double **coords) {
  // K, det = Compute Jacobian (coords)
  // Declaration of basis function matrices

  for (int i = 0; i<3; i++) {
    // Do generalized loop-invariant code motion
    for (int j = 0; j<4; j+=4)
      for (int k = 0; k<4; k+=4) {
        // load and set intrinsics
        // Compute A[0,0], A[1,1], A[2,2], A[3,3]
        // One permute_pd intrinsic per k-loop load
        // Compute A[0,1], A[1,0], A[2,3], A[3,2]
        // One permute2f128_pd intrinsic per k-loop load
        // ...
      }
    // Remainder loop (from j = 4 to j = 6)
  }
  // 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<4; 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 A[j+1][k], A[j+2][k], A[j+3][k]
    }
  }
}
```

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. Tiling at the level of vector registers represents another opportunity. 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

$$A_{jk} = \sum_{\substack{x \in B' \subseteq B \\ y \in B'' \subseteq B}} x_j \cdot y_k \qquad j,k = 0,...,2 \tag{5.1}$$

where B is the set of all basis functions (or temporary variables, e.g., `LI_0`)

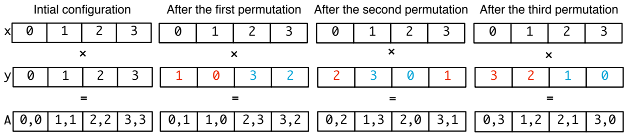
Intial configuration
After the first permutation
After the second permutation
After the third permutation
x
y
A
0 1 2 3
1 0 3 2
2 3 0 1
3 2 1 0
0,0 1,1 2,2 3,3
0,1 1,0 2,3 3,2
0,2 1,3 2,0 3,1
0,3 1,2 2,1 3,0

Figure 5.1: Outer-product vectorization by permuting values in a vector register.

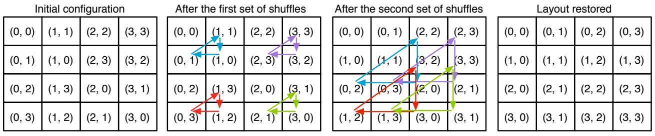
Initial configuration
After the first set of shuffles
After the second set of shuffles
Layout restored

Figure 5.2: 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.

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 5.1 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 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 5.2, 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 **??**.

5.2.4 Vector-expansion of Invariant Sub-expressions

...

5.3 Experiments

5.3.1 Experimental Setup

Experiments were run on a single core of an Intel architecture, a Sandy Bridge I7-2600 CPU running at 3.4GHz, with 32KB of L1 cache and 256KB of L2 cache). The `icc 14.1` compiler was used. On the Sandy Bridge, the compilation flags used were `-O2` and `-xAVX` for auto-vectorization. Other optimization levels performed, in general, slightly worse.

Three studies are presented. Firstly, in Section **??**, the impact of individual code transformations is analyzed in three real-world, representative equations. We believe it is important to clarify *in which contexts* and *why* such optimizations can provide speed ups.

...

5.3.2 Rationale of the Study

We evaluate some critical code transformations in three real-world problems based on the following PDEs: (i) Helmholtz, (ii) Diffusion, and (iii) Burgers. The evaluation concerns:

- Generalized loop-invariant code motion (Section **??**)
- Padding and data alignment (Section 5.2.1)
- Expression splitting (Section 5.2.2)
- Vector-register tiling (Section 5.2.3)

The goal of this section is to demonstrate the effectiveness of individual transformations and to explain under what circumnstances performance improvements are obtainable. It is important to understand *for what reasons* gains in runtime are achieved with these critical optimizations; some transformations are excluded from this study for reasons of time and readability of the section, but analogous considerations and observations would apply.

The three chosen equations are *real-life kernels* and comprise the core differential operators in some of the most frequently encountered finite element problems in scientific computing. This is of crucial importance because distinct problems, possibly arising in completely different fields, may employ (subsets of) the same differential operators of our benchmarks, which implies similarities and redundant patterns in the generated code. Consequently, the proposed code transformations have a domain of applicability that goes far beyond that of the three analyzed equations.

The Helmholtz and Diffusion kernels are archetypal second order elliptic operators. They are complete and unsimplified examples of the operators used to model diffusion and viscosity in fluids, and for imposing pressure in compressible fluids. As such, they are both extensively used in climate and ocean modeling. Very similar operators, for which the same optimisations are expected to be equally effective, apply to elasticity problems, which are at the base of computational structural mechanics. The Burgers kernel is a typical example of a first order hyperbolic conservation law, which occurs in real applications whenever a quantity is transported by a fluid (the momentum itself, in our case). We chose this particular kernel since it applies to a vector-valued quantity, while the elliptic operators apply to scalar quantities; this impacts the generated code, as explained next. The operators we have selected are characteristic of both the second and first order operators that dominate fluids and solids simulations.

The benchmarks were written in UFL (code available at [Luporini, 2014d]) and executed over real unstructured meshes through Firedrake. The Helmholtz code has already been shown in Listing 3. The Diffusion equation uses the same differential operators as Helmholtz. In the Diffusion kernel code, the main differences with respect to Helmholtz are the absence of the Y array and the presence of additional constants for computing the element matrix. Burgers is a non-linear problem employing differential operators different from those of Helmholtz and relying on vector-valued quantities, which has a major impact on the generated assembly code (see Listing 4), where a larger number of basis function arrays ($X1$, $X2$, ...) and constants ($F0$, $F1$, ..., $K0$, $K1$, ...) are generated.

These problems were studied varying both the shape of mesh elements and the polynomial order q of the method, whereas the element family, Lagrange, is fixed. As might be expected, the larger the element shape and

q, the larger the iteration space. Triangles, tetrahedra, and prisms were tested as element shape. For instance, in the case of Helmholtz with $q = 1$, the size of the `j` and `k` loops for the three element shapes is, respectively, 3, 4, and 6. Moving to bigger shapes has the effect of increasing the number of basis function arrays, since, intuitively, the behaviour of the equation has to be approximated also along a third axis. On the other hand, the polynomial order affects only the problem size (the three loops `i`, `j`, and `k`, and, as a consequence, the size of X and Y arrays). A range of polynomial orders from $q = 1$ to $q = 4$ were tested; higher polynomial orders are excluded from the study because of current Firedrake limitations. In all these cases, the size of the element matrix rarely exceeds 30×30, with a peak of 105×105 in Burgers with prisms and $q = 4$.

The speed ups due to applying optimizations over the original local assembly code are shown in Figure 5.3. The figure itself can be read as a "plot of plots": the various element shapes are reported on the horizontal axis, whereas each equation corresponds to one point along the vertical axis. This allows to deduct the behaviour of transformations when varying the parameters of our study in an unified way. In the next sections, we refer to this figure and elaborate on the impact of individual transformations. We shorten generalized loop-invariant code motion as *licm*; padding and data alignment as *ap*; outer-product vectorization as *op-vect*; expression splitting as *split*.

5.3.3 Impact of Transformations

Impact of Generalized Loop-invariant Code Motion

In general, the speedups achieved by *licm* are notable. The main reasons were anticipated in Section **??**: in the original code, 1) sub-expressions invariant to outer loops are not automatically hoisted, while 2) sub-expressions invariant to the innermost loop are hoisted, but their execution is not auto-vectorized. These observations come from inspection of assembly code generated by the compiler.

The gain tends to grow with the computational cost of the kernels: bigger loop nests (i.e., larger element shapes and polynomial orders) usually benefit from the reduction in redundant computation, even though extra memory for the temporary arrays is required. Some discrepancies to this

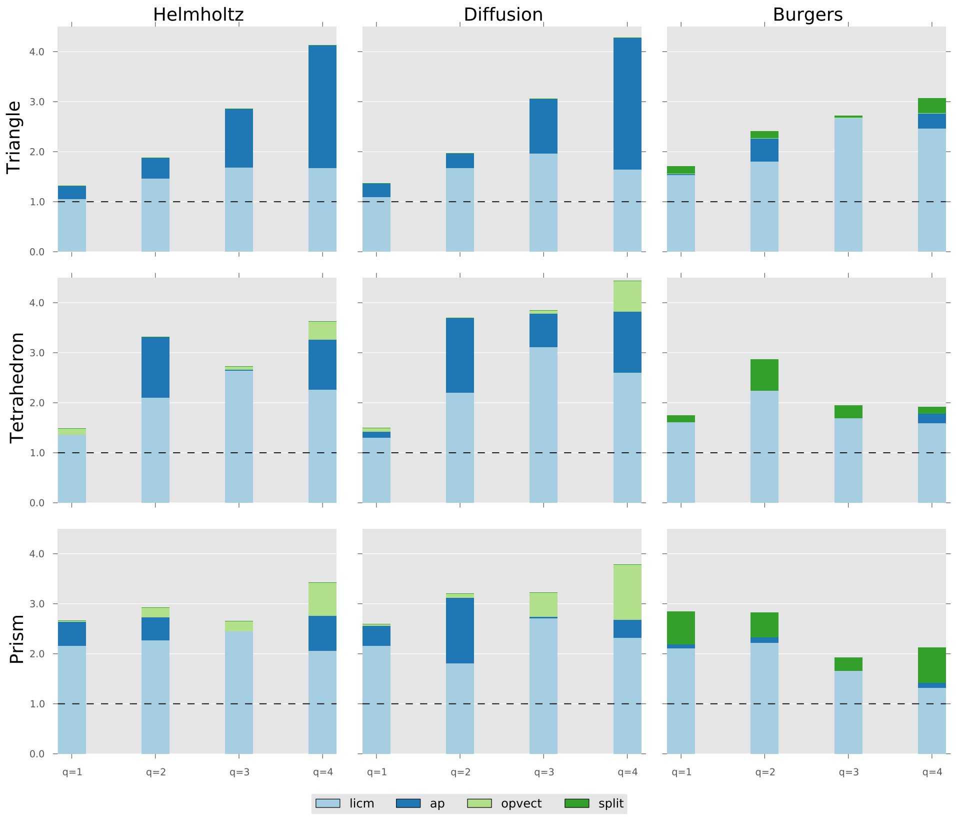

Figure 5.3: Performance improvement due to generalized loop-invariant code motion (*licm*), data alignment and padding (*ap*), outer-product vectorization (*op-vect*), and expression splitting (*split*) over the original non-optimized code. In each plot, the horizontal axis reports speed ups, whereas the polynomial order q of the method varies along the vertical axis.

trend are due to a less effective auto-vectorization. For instance, on the Sandy Bridge, the improvement at $q = 3$ is larger than that at $q = 4$ because, in the latter case, the size of the innermost loop is not a multiple of the vector length, and a remainder scalar loop is introduced at compile time. Since the loop nest is small, the cost of executing the extra scalar iterations can have a significant impact.

Impact of Padding and Data Alignment

Padding, which avoids the introduction of a remainder loop as described in Section 5.2.1, as well as data alignment, enhance the quality of auto-vectorization. Occasionally the impact of *ap* is marginal. These may be due to two reasons: (i) the non-padded element matrix size is already a multiple of the vector length; (ii) the number of aligned temporaries introduced by *licm* is so large to induce cache associativity conflicts (e.g. Burgers equation).

Impact of Vector-register Tiling

In this section, we evaluate the impact of vector-register tiling. *op-vect* requires the unroll-and-jam factor to be explicitly set. Here, we report the best speed-up obtained after all feasible unroll-and-jam factors were tried.

The rationale behind these results is that the effect of *op-vect* is significant in problems in which the assembly loop nest is relatively big. When the loops are short, since the number of arrays accessed at every loop iteration is rather small (between 4 and 8 temporaries, plus the element matrix itself), there is no need for vector-register tiling; extensive unrolling is sufficient to improve register re-use and, therefore, to maximize the performance. However, as the iteration space becomes larger, *op-vect* leads to improvements up to 1.4× (Diffusion, prismatic mesh, $q = 4$ - increasing the overall speed up from 2.69× to 3.87×).

Using the Intel Architecture Code Analyzer tool Intel Corporation [2012], we confirmed that speed ups are a consequence of increased register re-use. In Helmholtz $q = 4$, for example, the tool showed that when using *op-vect* the number of clock cycles to execute one iteration of the `j` loop decreases by roughly 17%, and that this is a result of the relieved pressure on both of the data (cache) ports available in the core.

The performance of individual kernels in terms of floating-point operations per second was also measured. The theoretical peak on a single core, with the Intel Turbo Boost technology activated, is 30.4 GFlop/s. In the case of Diffusion using a prismatic mesh and $q = 4$, we achieved a maximum of 21.9 GFlop/s with *op-vect* enabled, whereas 16.4 GFlop/s was obtained when only *licm-ap* is used. This result is in line with the expectations: analysis of assembly code showed that, in the `jk` loop nest, which in this problem represents the bulk of the computation, 73% of instructions are actually floating-point operations.

Application of *op-vect* to the Burgers problem induces significant slowdowns due to the large number of temporary arrays that need to be tiled, which exceeds the available logical registers on the underlying architecture. Expression splitting can be used in combination with *op-vect* to alleviate this issue; this is discussed in the next section.

Impact of Expression Splitting

Expression splitting relieves the register pressure when the element matrix evaluation needs to read from a large number of basis function arrays. As detailed in Section 5.2.2, the price to pay for this optimazion is an increased number of accesses to the element matrix and, potentially, redundant computation.

For the Helmholtz and Diffusion kernels, in which only between 4 and 8 temporaries are read at every loop iteration, `split` tends to slow down the computation, because of the aforementioned drawbacks. Slow downs up to 1.4× were observed.

In the Burgers kernels, between 12 and 24 temporaries are accessed at every loop iteration, so *split* plays a key role since the number of available logical registers on the Sandy Bridge architecture is only 16. In almost all cases, a split factor of 1, meaning that the original expression was divided into two parts, ensured close-to-peak perforance. The transformation negligibly affected register locality, so speed ups up to 1.5× were observed. For instance, when $q = 4$ and a prismatic mesh is employed, the overall performance improvement increases from 1.44× to 2.11×.

The performance of the Burgers kernel on a prismatic mesh was 20.0 GFlop/s from $q = 1$ to $q = 3$, while it was 21.3 GFlop/s in the case of $q = 4$. These values are notably close to the peak performance of 30.4 GFlop/s. Disabling *split* makes the performance drop to 17.0 GFlop/s for $q = 1, 2$, 18.2 GFlop/s for $q = 3$, and 14.3 GFlop/s for $q = 4$. These values are in line with the speedups shown in Figure 5.3.

The *split* transformation was also tried in combination with *op-vect* (*split-op-vect*). Despite improvements up to 1.22×, *split-op-vect* never outperforms *split*. This is motivated by two factors: for small split factors, such as 1 and 2, the data space to be tiled is still too big, and register spilling affects run-time; for higher ones, sub-expressions become so small that, as explained in Section 5.3.3, extensive unrolling already allows to achieve a certain degree of register re-use.

Results on Other Architectures

To avoid impairing the readability of the section, which already contains a vast amount of information, we are not reporting performance results on the Intel Xeon Phi architecture (each core running at 1.05Ghz in native mode, 32KB L1 cache and 512KB L2 cache). With respect to the Sandy Bridge, this architecture has a different number of logical registers and SIMD lanes (16 256-bit registers in the Sandy Bridge, and 32 512-bit registers in the Xeon Phi), which can impact the optimization strategy. The interested reader is invited to refer to the published paper [Luporini et al., 2014].

5.4 Experience with General-purpose Compiler Optimizations

5.4.1 Loop Interchange

All loops are interchangeable, provided that temporaries are introduced if the nest is not perfect. For the employed storage layout, the loop permutations `ijk` and `ikj` are likely to maximize performance. Conceptually, this is motivated by the fact that if the `i` loop were in an inner position, then a significantly higher number of load instructions would be required at every iteration. We tested this hypothesis in manually crafted kernels. We found that the performance loss is greater than the gain due to the possibility of accumulating increments in a register, rather than memory, along the `i` loop. The choice between `ijk` and `ikj` depends on the number of load instructions that can be hoisted out of the innermost dimension. As discussed in the next sections, a good heuristics it to choose as outermost the loop along which the number of invariant loads is smaller so that more registers remain available to carry out the computation of the local element matrix.

5.4.2 Loop Unroll

Loop unroll (or unroll-and-jam of outer loops) is fundamental to the exposure of instruction-level parallelism, and tuning unroll factors is particularly important.

We first observe that manual full (or extensive) unrolling is unlikely to be effective for two reasons. Firstly, the `ijk` loop nest would need to be small enough such that the unrolled instructions do not exceed the instruction cache, which is rarely the case: it is true that in a local assembly kernel the minimum size of the `ijk` loop nest is $3 \times 3 \times 3$ (triangular mesh and polynomial order 1), but this increases rapidly with the polynomial order of the method and the discretization employed (e.g. tetrahedral meshes imply larger loop nests than triangular ones), so sizes greater than $10 \times 10 \times 10$, for which extensive unrolling would already be harmful, are in practice very common. Secondly, manual unrolling is dangerous because it may compromise compiler auto-vectorization by either removing loops (most compilers search for vectorizable loops) or losing spatial locality within a vector register.

By comparison with implementations characterized by manually-unrolled loops, we noticed that recent versions of compilers like GNU's and Intel's estimate close-to-optimal unroll factors when the loops are affine and their bounds are relatively small and known at compile-time, which is the case of our kernels. Unless an auto-tuning system is used (this option is discussed in Section **??**) our choice is to leave the backend compiler in charge of applying loop unroll.

5.5 Related Work

The finite element method is extensively used to approximate solutions of PDEs. Well-known frameworks and applications include nek5000 [Paul F. Fischer and Kerkemeier, 2008], the FEniCS project [Logg et al., 2012], Fluidity [AMCG, 2010], and of course Firedrake. Numerical integration based on quadrature, as in Firedrake, is usually employed to implement the local assembly phase. The recent introduction of domain specific languages (DSLs) to decouple the finite element specification from its underlying implementation facilitated, however, the development of novel approaches. Methods based on tensor contraction [Kirby and Logg, 2006] and symbolic manipulation [Russell and Kelly, 2013] have been implemented. Nevertheless, it has been demonstrated that quadrature-based integration remains the most efficient choice for a wide class of problems [Olgaard and Wells, 2010], which motivates our work in COFFEE.

Optimization of quadrature-based local assembly for CPU architectures has been addressed in FEniCS Olgaard and Wells [2010]. The comparison between COFFEE and this work has been presented in Section **??**. In [Markall et al., 2010], and more recently in [Knepley and Terrel, 2013], the same problem has been studied for GPU architectures. In [Kruel and Bana, 2013], variants of the standard numerical integration algorithm have been specialized and evaluated for the PowerXCell processor, but an exhaustive study from the compiler viewpoint - like ours - is missing, and none of the optimizations presented in this chapter are mentioned. Among these efforts, to the best of our knowledge, COFFEE is the first work targeting low-level optimizations through a real compiler approach.

The code transformations presented are inspired by standard compilers optimizations and exploit domain properties. Our loop-invariant code motion technique individuates invariant sub-expressions and redundant computation by analyzing all loops in an iteration space, which is a generalization of the algorithms often implemented by general-purpose compilers. Expression splitting is an abstract variant of loop fission based on properties of arithmetic operators. The outer-product vectorization is an implementation of tiling at the level of vector registers; tiling, or "loop blocking", is commonly used to improve data locality (especially for caches). Padding has been used to achieve data alignment and to improve the effectiveness of vectorization. A standard reference for the compilation techniques re-adapted in this work is [Aho et al., 2007].

Our compiler-based optimization approach is made possible by the top-level DSL, which enables automated code generation. DSLs have been proven successful in auto-generating optimized code for other domains: Spiral [Püschel et al., 2005] for digital signal processing numerical algorithms, [Spampinato and Püschel, 2014] for dense linear algebra, or Pochoir [Tang et al., 2011] and SDSL [Henretty et al., 2013] for image processing and finite difference stencils. Similarly, PyOP2 is used by Firedrake to express iteration over unstructured meshes in scientific codes.

COFFEE improves automated code generation in Firedrake.

Many code generators, like those based on the Polyhedral model [Bondhugula et al., 2008] and those driven by domain-knowledge [Stock et al., 2011], make use of cost models. The alternative of using auto-tuning to select the best implementation for a given problem on a certain platform has been adopted by nek5000 [Shin et al., 2010] for small matrix-matrix multiplies, the ATLAS library [Whaley and Dongarra, 1998], and FFTW [Frigo and Johnson, 2005] for fast fourier transforms. In both cases, pruning the implementation space is fundamental to mitigate complexity and overhead. Likewise, COFFEE uses heuristics and a model-driven auto-tuning system (Section **??**) to steer the optimization process.

5.6 Generality of the Approach and Applicability to Other Domains

We have demonstrated that our cross-loop optimizations for arithmetic intensity are effective in the context of automated code generation for finite element local assembly. In this section, we discuss their applicability in other computational domains and, in general, their integrability within a general-purpose compiler.

COFFEE was developed as a separate, self-contained software module, with clear input/output interfaces, rather then incorporating it within PyOP2. This choice was motivated by two critical aspects that characterize the generality of our research:

Separation of concerns We believe that in domain-specific frameworks there must be a clear, logical separation of roles reflecting the various levels of abstraction, where domain specialists are completely separated from performance optimization. In Firedrake, for instance, COFFEE decouples the mathematical specification of a finite element method, captured by the Unified Form Language and the FEniCS Form Compiler, from code optimization. This is of fundamental importance to maximize productivity by allowing scientists to focus only on their area of expertise. Practically speaking, from the perspective of the domain-specific language and compiler designers, our optimization strategy represents an incentive to produce extremely simple representations of the code (e.g. fully-inlined mathematical expressions in the form of an abstract syntax tree, in the case of Firedrake) so as to make the architecture-aware code optimizer completely responsible for choosing and applying the most suitable set of transformations.

Generalizability to other domains There are neither conceptual nor technical reasons which prevent our compiler from being used in applications other than Firedrake. For example, integration with the popular FEniCS framework, the pioneer of automated code generation for finite element local assembly, would be relatively easy to achieve. It is more challenging to assess the generality of the optimization strategy: the extent to which COFFEE and its transformations are transferable to other computational domains, perhaps other DSLs, and to what extent this would be helpful for improving full application performance. To answer these questions, we first need to go back to the origins of our compiler. The starting point of our work was the mathematical formulation of a local assembly operation, expressible as follows

$$\forall_{i,j} \quad A_{ij}^{K} = \sum_{q=1}^{n_1} \sum_{k=1}^{n_2} \alpha_{k,q}(a',b',c',...)\beta_{q,i,j}(a,b,c,d,...)\gamma_q(w_K,z_K) \tag{5.2}$$

The expression represents the numerical evaluation of an integral at n_1 points in the mesh element K computing the local element matrix A. Functions α, β and γ are problem-specific and can be intricately complex, involving for example the evaluation of derivatives. We can however abstract from the inherent structure of α, β and γ to highlight a number of aspects

- **Optimizing mathematical expressions.** Expression manipulation (e.g. simplification, decomposition into sub-expressions) opens multiple semantically equivalent code generation opportunities, characterized by different trade-offs in parallelism, redundant computation, and data locality. The basic idea is to exploit properties of arithmetic operators, such as associativity and commutativity, to re-schedule the computation suitably for the underlying architecture. Loop-invariant code motion and expression splitting follow this principle, so they can be re-adapted or extended to any domains involving numerical evaluation of complex mathematical expressions (e.g. electronic structure calculations in physics and quantum chemistry relying on tensor contractions Hartono et al. [2009]). In this context, we highlight three notable points.

1. In Equation (5.2), the summations correspond to reduction loops, whereas loops over indices i and j are fully parallel. Throughout the paper we assumed that a kernel will be executed by a single thread, which is likely to be the best strategy for standard multi-core CPUs. On the other hand, we note that for certain architectures (for example GPUs) this could be prohibitive due to memory requirements. Intra-kernel parallelization is one possible solution: a domain-specific compiler such as COFFEE could map mathematical quantifiers and operators to different parallelization schemes and generate distinct variants of multi-threaded kernel code. Based on our experience, we believe this is the right approach to achieve performance portability.
2. The various sub-expressions in β only depend on (i.e. iterate along) a subset of the enclosing loops. In addition, some of these sub-expressions might reduce to the same values as iterating along certain iteration spaces. This code structure motivated the generalized loop-invariant code motion technique. The intuition is that whenever sub-expressions invariant with respect to different sets of affine loops can be identified, the question of whether, where and how to hoist them, while minimizing redundant computation, arises. Pre-computation of invariant terms also increases memory requirements due to the need for temporary arrays, so it is possible that for certain architectures the transformation could actually cause slowdowns (e.g. whenever the available per-core memory is small).
3. Associative arithmetic operators are the prerequisite for expression splitting. In essence, this transformation concerns resource-aware execution. In the context of COFFEE, expression splitting has been successfully applied to improve register pressure. However, the underlying idea of re-scheduling (re-associating) operations to optimize for some generic parameters is far more

general. It could be used, for example, as a starting point to perform kernel fission; that is, splitting a kernel into multiple parts, each part characterized by less stringent memory requirements (a variant of this idea for non-affine loops in unstructured mesh applications has been adopted in [Bertolli et al., 2013]). In Equation (5.2), for instance, not only can any of the functions α, β and γ be split (assuming they include associative operators), but α could be completely extracted and evaluated in a separate kernel. This would reduce the working set size of each of the kernel functions, an option which is particularly attractive for many-core architectures in which the available per-core memory is much smaller than that in traditional CPUs.

- **Code generation and applicability of the transformations.** All array sizes and loop bounds, for example $n1$ and $n2$ in Equation 5.2, are known at code generation time. This means that "good" code can be generated. For example, loop bounds can be made explicit, arrays can be statically initialized, and pointer aliasing is easily avoidable. Further, all of these factors contribute to the applicability and the effectiveness of some of our code transformations. For instance, knowing loop bounds allows both generation of correct code when applying vector-register tiling and discovery of redundant computation opportunities. Padding and data alignment are special cases, since they could be performed at run-time if some values were not known at code generation time. Theoretically, they could also be automated by a general-purpose compiler through profile-guided optimization, provided that some sort of data-flow analysis is performed to ensure that the extra loop iterations over the padded region do not affect the numerical results.

- **Multi-loop vectorization.** Compiler auto-vectorization has become increasingly effective in a variety of codes. However, to the best of our knowledge, multi-loop vectorization involving the loading and storing of data along a subset of the loops characterizing the iteration space (rather than just along the innermost loop), is not supported by available general-purpose and polyhedral compilers. The outer-product vectorization technique presented in this paper shows that two-loop vectorization can outperform standard auto-vectorization. In addition, we expect the performance gain to scale with the number of vectorized loops and the vector length (as demonstrated in the Xeon Phi experiments). Although the automation of multi-loop vectorization in a general-purpose compiler is far from straightforward, especially if stencils are present, we believe that this could be more easily achieved in specific domains. The intuition is to map the memory access pattern onto vector registers, and then to exploit in-register shuffling to minimize the traffic between memory and processor. By demonstrating the effectiveness of multi-loop vectorization in a real scenario, our research represents an incentive for studying this technique in a broader and systematic way.

5.7 Conclusion

In this chapter, we have presented the study and systematic performance evaluation of a class of composable cross-loop optimizations for improving arithmetic intensity in finite element local assembly kernels, and their integration in a novel compiler, COFFEE. In the context of automated code generation for finite element local assembly, COFFEE is the first compiler capable of introducing low-level optimizations to maximize instruction-level parallelism, register locality and SIMD vectorization. Assembly kernels have particular characteristics. Their iteration space is usually very small, with the size depending on aspects like the degree of accuracy one wants to reach (polynomial order of the method) and the mesh discretization employed. The data space, in terms of number of arrays and scalars required to evaluate the element matrix, grows proportionally with the complexity of the finite element problem. COFFEE has been developed taking into account all of these degrees of freedom, based on the idea that reducing the problem of local assembly optimization to a fixed sequence of transformations is far too superficial if close-to-peak performance needs to be reached. The various optimizations overcome limitations of current vendor and research compilers. The exploitation of domain knowledge allows some of them to be particularly effective, as demonstrated by our experiments on a state-of-the-art Intel platform. COFFEE supports all of the problems expressible in Firedrake, and is integrated with this framework, which demonstrate the maturity of the research performed. The generality and the applicability of the proposed code transformations to other domains has also been discussed.

Chapter 6

COFFEE: a Compiler for Fast Expression Evaluation

6.1 Overview

...

6.2 Outline and Integration with Firedrake

Our research resulted in the implementation of COFFEE[1], a mature, platform-independent compiler capable of optimizing any local assembly code generated through Firedrake or FEniCS. COFFEE has been fully integrated with Firedrake, so it can optimize any equation expressible with this framework. The code, which comprises more than 5000 lines, is available at [Luporini, 2014a].

Firedrake users employ the Unified Form Language to express problems in a notation resembling mathematical equations. At run-time, the high-level specification is translated by a modified version of its form compiler, the FEniCS Form Compiler (FFC) Kirby and Logg [2006], into an abstract syntax tree (AST) representation of one or more finite element assembly kernels. ASTs are then passed to COFFEE to apply the transformations described in the previous sections. The output of COFFEE, C code, is eventually provided to PyOP2 [Markall et al., 2013], where just-

---

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

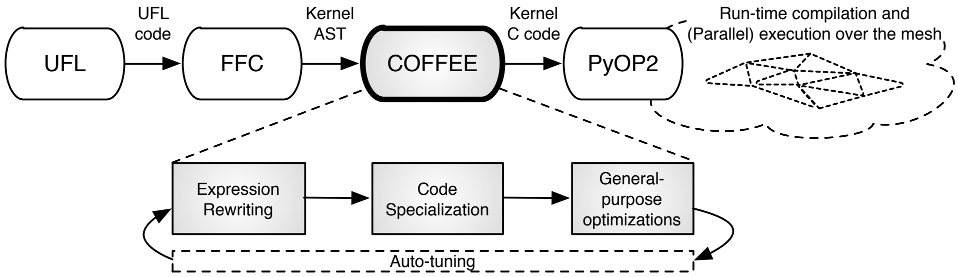
UFL
UFL code
FFC
Kernel AST
COFFEE
Kernel C code
PyOP2
Run-time compilation and (Parallel) execution over the mesh
Expression Rewriting
Code Specialization
General-purpose optimizations
Auto-tuning

Figure 6.1: High-level view of Firedrake. COFFEE is at the core, receiving ASTs from a modified version of the FEniCS Form Compiler and producing optimized C code kernels.

in-time compilation and execution over the discretized domain take place. The flow and the compiler structure are outlined in Figure 6.1.

The compiler applies an ordered sequence of optimization steps to the ASTs received from FFC. Expression rewriting is performed first, due to the introduction of temporary arrays and the need to create multiple loop nest starting from the original one (this may happen when computation over zero-valued blocks is avoided). Then, code specialization takes place. Finally, loop interchange and loop unroll are applied. COFFEE has an auto-tuning system, described in Section **??**, to individuate the most suitable optimization strategy (i.e. to select which transformations to apply) for a certain equation. The implementation of all code transformations is centered on analysis and manipulation of the kernel AST. Any possible corner cases are handled: for example, if outer-product vectorization is to be applied but the size of the iteration space is not a multiple of the vector length, then a remainder loop, amenable to auto-vectorization, is inserted (as shown in Figure **??**).

Users can either enable individual transformations by switching specific flags or leave the auto-tuning system, described in Section **??**, in charge of determining the best optimization strategy (i.e. selecting and composing transformations).

6.3 Modifying the FEniCS Form Compiler

COFFEE expects as input either a string of C code or an AST. In the former case, a parser could be used to obtain an AST representation, which is required by the algorithms implementing the various code transformations. Therefore, FFC's output, which is a C-code implementation of a local assembly kernel, could be used straightforwardly as input to COFFEE. However, we note that this would also be conceptually pointless: FFC would generate C code from its intermediate representation that eventually COFFEE would have to re-parse to obtain an AST. A much cleaner solution, adopted and explained next, consists of modifying FFC to directly generate an AST.

6.3.1 Generation of Abstract Syntax Trees

The construction of the FFC's intermediate representation from UFL code is refined as follows:

- The mathematical expression that evaluates the element matrix is represented by a tree data structure. A limitation of the original FFC was that nodes in such expression tree, which correspond to symbols or arithmetic operations, are not bound to the enclosing loops. For instance, consider the symbol `A[i][j]`: the FFC's expression tree has a node for this symbol, but visiting it there is no clean way of separating the variable name `A` from the loop indices `i` and `j`. Therefore, we have enriched symbol nodes with additional fields to capture these information.

- Basis functions in the FFC's intermediate representation are characterized by a new field telling whether they originated from a vector-valued or a scalar-valued element. In the former case, the array rappresenting the tabulation of a basis function at the various quadrature points is block-sparse. This information is recorded and, as explained next, attached to the kernel's AST to enable COFFEE applying symbolic execution to avoid iteration over zero-valued blocks, as elaborated in Section **??**.

In the modified FFC, the intermediate representation is intercepted prior to the code generation stage and forwarded to a new module that builds

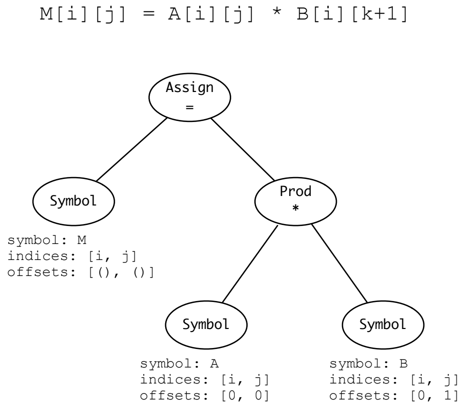

Figure 6.2: AST representation of a C assignment in COFFEE.

ASTs. To this purpose, COFFEE exposes its hierarchy of AST nodes, whose design strictly adhere to standard object-oriented programming rules (e.g. by making extensive use of overloading). An example of such an AST representation for a simple C statement is provided in Figure 6.2.

The enriched intermediate representation allows to create proper AST's Symbol nodes. A Symbol, as shown in the figure, has a name (the variable name), an array of indices, and an array of offsets. These information are fundamental for the implementation of expression rewriting and code specialization. Further, special AST's Declaration objects can be used to track the sparsity pattern in basis function arrays.

6.4 Implementation

Figure 6.3 outlines the various modules composing COFFEE. Expression rewriting, code specialization, and general-purpose transformations are applied by manipulating the AST representing the local assembly kernel. Providing the steps of all algorithms performing the various transformations would just be tedious and marginally helpful for the reader; essentially, implementing a transformation always reduces to write a routine that visits and manipulates a tree data structure (the AST) according to the semantics described in Sections **??** and **??**. In this section, we rather focus on aspects of the compiler implementation that involve the orchestration

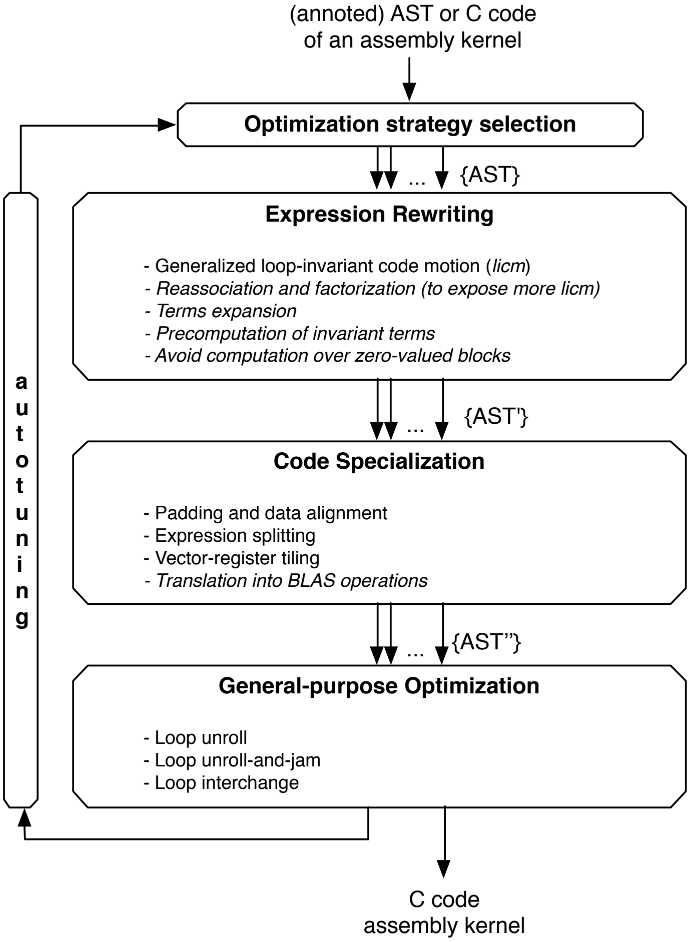
(annoted) AST or C code
of an assembly kernel
Optimization strategy selection
... {AST}
Expression Rewriting
- Generalized loop-invariant code motion (licm)
- Reassociation and factorization (to expose more licm)
- Terms expansion
- Precomputation of invariant terms
- Avoid computation over zero-valued blocks
... {AST'}
Code Specialization
- Padding and data alignment
- Expression splitting
- Vector-register tiling
- Translation into BLAS operations
... {AST''}
General-purpose Optimization
- Loop unroll
- Loop unroll-and-jam
- Loop interchange
autotuning
C code
assembly kernel

Figure 6.3: Structure of COFFEE.

of the different transformations, including a description of the data structures employed to track both the restructuring of the assembly expressions and data dependencies.

6.4.1 Code Hoisting

Hoisting is the operation of moving code, for instance a statement or the evaluation of an expression, from an inner to an outer loop in a nest. This operation is performed in several occasions when rewriting an expression: generalized loop-invariant code motion, expansion of sub-expressions, and precomputation of terms all require code hoisting, as illustrated in Figures **??** and **??**. In order to perform code hoisting, in particular, several aspects must be known:

1. "what" to hoist
2. "where" to hoist (possibly, outside of the loop nest)
3. if scalar-expansion should be introduced

Answers to these three points obviously depend on the specific transformation, although parts of the implementation are shared. COFFEE tracks hoisted code by means of a dictionary that binds variable names (guaranteed to be unique) to a set of information. In particular, for a variable `v`, the dictionary provides:

- a reference to the AST node corresponding to the expression `e` hosted by `v`;
- a reference to the loop in which `v` is assigned `e`;
- a reference to the AST node corresponding to the declaration of `v`.

As expressions are hoisted, the dictionary is populated and/or updated with new information. For example, when applying generalized loop-invariant code motion, a new entry is created, unless the sub-expression being lifted has already been pre-computed elsewhere. On the other hand, when sub-expressions are expanded, either a new entry is created or an old entry is updated, as explained in Section **??**.

The dictionary is also queried at code specialization time for padding, data alignment and generation of BLAS calls. Therefore, different algorithms in COFFEE access the same data structure, which allows avoiding both duplicated code and an additional overhead due to revisiting the same portion of AST in distinct transformations.

6.4.2 Benefits of Tracking Data Dependencies

COFFEE implements "smart" code hoisting: everytime a sub-expression or a term are lifted, for example from the mathematical expression evaluating the local element matrix to an outer level in the loop nest, three optimizations are potentially applied. Consider a hoistable expression `e` assuming different values in the iteration space `I`:

- **Minimize redundant computation.** If an equivalent sub-expression `e'` has already been hoisted along the iteration space `I`, then COFFEE replaces `e` with a reference to the symbol hosting the value of `e'`. This avoids both redundant computation and the introduction of additional temporary variables;

- **Loop fusion.** If scalar-expansion is introduced (we recall this is useful to achieve SIMD auto-vectorization; see Sections **??** and 5.2.4), then the hoisted code must be placed in an outer loop `r`, $r \in I$. This was the purpose of the second `r` loop in Figure **??**; note that this loop includes scalar-expanded sub-expressions that, in the non-transformed code, iterated along logically different spaces (loops `j` and `k`, corresponding to test and trial functions). In this example, it was possible to use a single `r` loop because we assumed that test and trial function spaces were the same, leading to identical loop bounds; in general, however, this is not true. Another assumption of the example was that the space of the equation's coefficients (variables `f0, f1` in the figure) coincided with that of test and trial functions. This would allow fusing the two `r` loops, which is exactly what COFFEE does, although not displayed by the figure. Fusing loops, which increases data locality and reduces loop overhead, is possible in some circumnstances, in particular when there are no data dependencies among hoisted expressions and the spaces of test, trial, and coefficent functions are identical. As explained next, COFFEE reasons about data dependencies and iteration spaces to determine the safeness of loop fusion.

- **Reduce extra memory.** When expanding an expression, terms hoisting is possible to relieve register pressure. This poses the challenge described in Section **??** and intuitively summarized in Figure **??**; that is, understanding whether it is possible to absorb the hoistable term in an available temporary value or a new temporary is needed. Obviously, the less is the number of temporaries introduced, the smaller is the size of the working set, which may result in better performance.

To implement these optimizations, it is necessary to track the evolution of data dependencies as the assembly expression is rewritten. For this purpose, COFFEE uses a dependency graph, which is a standard approach used by general-purpose compilers relying on abstract syntax trees (in addition to other data structures) as intermediate representation. The dependency graph has as many nodes as the number of variables in the loop nest characterizing the assembly expression. A direct edge from a node `A` to a node `B` indicates that the value of symbol `B` depends on that of `A`.

The implementation of the depedency graph data structure and of the various algorithms using it is made simple by the fact that COFFEE ensures *static single assignment* form. This property, typically adopted by intermediate representations in compilers, requires that variables are assigned exactly once, and that each variable is defined in advance.

Chapter 7

Conclusions

...

Bibliography

M. F. Adams and J. Demmel. Parallel multigrid solver algorithms and implementations for 3D unstructured finite element problem. In *Proceedings of SC99*, Portland, Oregon, November 1999.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007. ISBN 0-321-48681-1. URL `http://www.loc.gov/catdir/toc/ecip0618/2006024333.html`.

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`.

Saman Amarasinghe, Mary Hall, Richard Lethin, Keshav Pingali, Dan Quinlan, Vivek Sarkar, John Shalf, Robert Lucas, Katherine Yelick, Pavan Balanji, Pedro C. Diniz, Alice Koniges, and Marc Snir. Exascale programming challenges. In *Proc. Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA*. ASCR, DOE, Jul 2011.

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`.

Ayon Basumallik and Rudolf Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proc. 11th ACM SIGPLAN Symp. Prin. & Prac. of Par. Prog.*, pages 119–128, New York, New York, USA, 2006. ACM.

C. Bertolli, A. Betts, N. Loriant, G.R. Mudalige, D. Radford, D.A. Ham, M.B. Giles, and P.H.J. Kelly. Compiler optimizations for industrial unstructured mesh cfd applications on gpus. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, volume 7760 of *Lecture Notes in Computer Science*, pages 112–126. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37657-3. doi: 10.1007/978-3-642-37658-0_8.

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`.

T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1 – 1:25, 2011a.

Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011b.

James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2008.

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`.

Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis*, pages 21–40, February 2000.

Denys Dutykh, Raphaël Poncet, and Frédéric Dias. The VOLNA code for the numerical modeling of tsunami waves: Generation, propagation and inundation. *European Journal of Mechanics - B/Fluids*, 30(6):598 – 615, 2011. Special Issue: Nearshore Hydrodynamics.

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

Matteo Frigo and Steven G. Johnson. The design and implementation of fftw3. In *Proceedings of the IEEE*, pages 216–231, 2005.

M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H.J. Kelly. Performance analysis of the op2 framework on many-core architectures. *SIGMETRICS Perform. Eval. Rev.*, 38(4):9–15, March 2011. ISSN 0163-5999. doi: 10.1145/1964218.1964221. URL `http://doi.acm.org/10.1145/1964218.1964221`.

MB Giles, D Ghate, and MC Duta. Using automatic differentiation for adjoint cfd code development. 2005.

A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D. E. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry†. *The Journal of Physical Chemistry A*, 113(45):12715–12723, 2009. doi: 10.1021/jp9051215. URL `http://pubs.acs.org/doi/abs/10.1021/jp9051215`. PMID: 19888780.

Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2130-3.

doi: 10.1145/2464996.2467268. URL `http://doi.acm.org/10.1145/2464996.2467268`.

Intel Corporation. *Intel architecture code analyzer (IACA)*, 2012. [Online]. Available: http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/.

George Karypis and Vipin Kumar. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 5.0. `http://www.cs.umn.edu/~metis`, 2011.

Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32(3):417–444, September 2006. ISSN 0098-3500. doi: 10.1145/1163641.1163644. URL `http://doi.acm.org/10.1145/1163641.1163644`.

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`.

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`.

Christopher D. Krieger and Michelle Mills Strout. Executing optimized irregular applications using task graphs within existing parallel models. In *Proceedings of the Second Workshop on Irregular Applications: Architectures and Algorithms (IA^3) held in conjunction with SC12*, November 11, 2012.

Christopher D. Krieger, Michelle Mills Strout, Catherine Olschanowsky, Andrew Stone, Stephen Guzik, Xinfeng Gao, Carlo Bertolli, Paul Kelly, Gihan Mudalige, Brian Van Straalen, and Sam Williams. Loop chaining: A programming abstraction for balancing locality and parallelism. In *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Boston, Massachusetts, USA, May 2013.

Filip Kruel and Krzysztof Bana. Vectorized opencl implementation of numerical integration for higher order finite elements. *Comput. Math. Appl.*, 66(10):2030–2044, December 2013. ISSN 0898-1221. doi: 10.1016/j.camwa.2013.08.026. URL `http://dx.doi.org/10.1016/j.camwa.2013.08.026`.

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.

Fabio Luporini. COFFEE code repository. `https://github.com/OP2/PyOP2/tree/master/pyop2/coffee`, 2014a.

Fabio Luporini. Code of experiments on forms of increasing complexity in COFFEE. `https://github.com/firedrakeproject/firedrake-bench/blob/experiments/forms/firedrake_forms.py`, 2014b.

Fabio Luporini. Code of the full application study in COFFEE. `https://github.com/firedrakeproject/firedrake-bench/tree/experiments/elasticity`, 2014c.

Fabio Luporini. Code of experiments on individual transformations in COFFEE. `https://github.com/firedrakeproject/firedrake/tree/pyop2-ir-perf-eval/tests/perf-eval`, 2014d.

Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. Cross-loop optimization of arithmetic intensity for finite element local assembly. 2014.

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.

Graham R. Markall, David A. Ham, and Paul H.J. Kelly. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science*, 1(1):1815 – 1823, 2010. ISSN 1877-0509. doi: http://dx.doi.org/10.1016/j.procs.2010.04.203. URL `http://www.`

sciencedirect.com/science/article/pii/S1877050910002048. ICCS 2010.

Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 36:1–36:12. ACM, 2009. ISBN 978-1-60558-744-8. doi: http://doi.acm.org/10.1145/1654059.1654096.

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.

James W. Lottes Paul F. Fischer and Stefan G. Kerkemeier. nek5000 Web page, 2008. http://nek5000.mcs.anl.gov.

Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *Proc. Intl. Conf. on High Perf. Comp., Net., Sto. & Anal.*, pages 72:1–72:11, 2012. ISBN 978-1-4673-0804-5.

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.

Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5): 603–612, 1991.

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`.

Daniele G. Spampinato and Markus Püschel. A basic linear algebra compiler. In *International Symposium on Code Generation and Optimization (CGO)*, 2014.

K. Stock, T. Henretty, I. Murugandi, P. Sadayappan, and R. Harrison. Model-driven simd code generation for a multi-resolution tensor kernel. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1058–1067, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4385-7. doi: 10.1109/IPDPS.2011.101. URL `http://dx.doi.org/10.1109/IPDPS.2011.101`.

Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, July 2002.

Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proc. ACM SIGPLAN Conf. Prog. Lang. Des. & Impl. (PLDI)*, New York, NY, USA, June 2003. ACM.

Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications*, 18(1):95–114, February 2004.

M.M. Strout, F. Luporini, C.D. Krieger, C. Bertolli, G.-T. Bercea, C. Olschanowsky, J. Ramanujam, and P.H.J. Kelly. Generalizing runtime tiling with the loop chain abstraction. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1136–1145, May 2014. doi: 10.1109/IPDPS.2014.118.

Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. doi: 10.1145/1989493.1989508. URL `http://doi.acm.org/10.1145/1989493.1989508`.

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`.

R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X. URL `http://dl.acm.org/citation.cfm?id=509058.509096`.