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 Abstractions in Computational Science 11

2.2.1 Domain Specific Languages 11

2.2.2 Multilayer Frameworks for the Finite Element Method 11

2.2.3 Abstractions for Mesh Iteration 11

2.3 Compilers and Libraries for Code Optimization 11

2.3.1 Loop Optimization - Static Analysis 11

2.3.2 Loop Optimization - Dynamic Analysis 11

2.3.3 On the Dichotomy between Loop Tiling and Loop Fusion 12

2.3.4 Domain Specific Optimization 12

2.4 State-of-the-art Hardware Architectures 12
  2.4.1 SIMD Vectorization 13
  2.4.2 Terminology 13

3 Automated Tiling for Irregular Computations 15
3.1 Motivation 15
3.2 Tackling Real-World Applications 16
  3.2.1 Mesh-based Numerical Methods for PDEs 16
  3.2.2 Hypotheses 18
  3.2.3 Automating Loop Tiling is More Difficult than Commonly Thought 19
3.3 Related Work 22
3.4 Generalized Inspector/Executor Schemes 23
  3.4.1 The Loop Chain Abstraction 24
  3.4.2 Loop Chains for Unstructured Mesh Computations . 25
3.5 Sparse Tiling Examples 27
  3.5.1 Overview of the Algorithm 27
  3.5.2 Shared-Memory Parallelism 28
  3.5.3 Distributed-Memory Parallelism 29
3.6 Formalization 32
  3.6.1 Data Dependency Analysis for Loop Chains 32
  3.6.2 The Generalized Sparse Tiling Algorithms 33
  3.6.3 Inspection 33
  3.6.4 Execution 35
  3.6.5 Correctness of Inspection 36
  3.6.6 Computational Complexity of Inspection 36
3.7 Automation 36
  3.7.1 SLOPE: a Library for Tiling Irregular Computations 36
  3.7.2 PyOP2: a Runtime Library for Mesh Iteration 36
  3.7.3 Firedrake/DMPlex: the S-depth mechanism for MPI 36
3.8 Performance Evaluation 36
  3.8.1 Benchmarks 36
  3.8.2 Seigen: an Elastic Wave Equation Solver for Seismological Problems 36

4 On Optimality of Finite Element Integration 37

5 **Cross-loop Optimization of Arithmetic Intensity for Finite Element Integration** **39**
- 5.1 Recapitulation and Objectives 39
- 5.2 Low-level Optimization 42
  - 5.2.1 Padding and Data Alignment 42
  - 5.2.2 Expression Splitting 43
  - 5.2.3 Model-driven Vector-register Tiling 45
- 5.3 Experiments 47
  - 5.3.1 Setup 47
  - 5.3.2 Impact of Transformations 49
- 5.4 Experience with Traditional Compiler Optimizations 53
  - 5.4.1 Loop Interchange 53
  - 5.4.2 Loop Unroll 53
  - 5.4.3 Vector promotion 54
  - 5.4.4 Loop Fusion 55
- 5.5 Related Work 55
- 5.6 Applicability to Other Domains 56
- 5.7 Conclusion 60

6 **COFFEE: a Compiler for Fast Expression Evaluation** **61**
- 6.1 Overview 61
- 6.2 The Compilation Pipeline 62
- 6.3 Plugging COFFEE into Firedrake 64
  - 6.3.1 Abstract Syntax Trees 64
  - 6.3.2 Integration with Form Compilers 65
- 6.4 Rewrite Operators 67
- 6.5 Features of the Implementation 68
  - 6.5.1 Tree Visitor Pattern 68
  - 6.5.2 Flexible Code Motion 69
  - 6.5.3 Tracking Data Dependency 70
  - 6.5.4 Minimizing Temporaries 71

7 **Conclusions** **73**

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. 47
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. 47
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. 49

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. 62
6.2 AST representation of a C assignment in COFFEE. 65

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.

```
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 6.3; 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 Abstractions in Computational Science

2.2.1 Domain Specific Languages

...

2.2.2 Multilayer Frameworks for the Finite Element Method

Firedrake and FEniCS

2.2.3 Abstractions for Mesh Iteration

Structured Meshes ...

Unstructured Meshes OP2, PyOP2, Lizst

OP2 and PyOP2

LISTING 5: ...

```
void kernel1 (double * x, double * tmp1, double * tmp2)  *tmp1 += *x; *tmp2 += *x;
// loop over edges op_par_loop (edges, kernel1, op_arg_dat (x, -1, OP_ID, OP_READ),
op_arg_dat (temp, 0, edges2vertices, OP_INC), op_arg_dat (temp, 1, edges2vertices,
OP_INC))
// loop over cells op_par_loop (cells, kernel2, op_arg_dat (temp, 0, cells2vertices,
OP_INC), op_arg_dat (temp, 1, cells2vertices, OP_INC), op_arg_dat (temp, 2,
cells2vertices, OP_INC), op_arg_dat (res, -1, OP_ID, OP_READ))
// loop over edges op_par_loop (edges, kernel3, op_arg_dat (temp, 0, edges2vertices,
OP_INC), op_arg_dat (temp, 1, edges2vertices, OP_INC))
```

2.3 Compilers and Libraries for Code Optimization

2.3.1 Loop Optimization - Static Analysis

Polyhedral compilers ... scheduling functions Mention their unsuitability for tiling unstructured meshes... (use email I sent listing all issues...)

2.3.2 Loop Optimization - Dynamic Analysis

- Inspector/Executor schemes
- Space filling curves for unstructured meshes
- sparse tiling = loop fusion + loop tiling

2.3.3 On the Dichotomy between Loop Tiling and Loop Fusion

- Tiling is for a nested loop !
- tiles are atomic blabla
- split vs overlapped

Split tiling

Overlapped tiling

Fusion vs Tiling

so time tiling is just fusion across the time stepping loop, then tiling

2.3.4 Domain Specific Optimization

...

Tensor Contraction Engine

...

LGen

Why potentially useful ...

Halide

...

Spiral

...

2.4 State-of-the-art Hardware Architectures

...

2.4.1 SIMD Vectorization

...

2.4.2 Terminology

Memory pressure, Register pressure

Arithmetic intensity

Operational intensity Maybe add a subsection of the roofline model ?

Flops

Access function (for array)

General-purpose compiler

Communication vs computation

CPU- vs Memory-bound

Local vs Global reduction

Chapter 3

Automated Tiling for Irregular Computations

3.1 Motivation

Many numerical methods for partial differential equations (PDEs) are structured as sequences of parallel loops, often interleaved by sparse linear algebra operations. This exposes parallelism well, but often does not convert data reuse between loops into data locality, since very large datasets are usually accessed. In Section 2.3.1 we have explained that loop fusion and loop tiling may be used to retain a significant fraction of this potential data locality. However, as elaborated in Section 3.2, the complexity inherent in real-world applications makes it challenging to adopt such optimizations in practice.

Our focus is on unstructured mesh PDE solvers, such as those based on the finite volume or the finite element methods. Here, the loop-to-loop dependence structure is data-dependent due to indirect references such as `A[map[i]]`; the `map` array stores connectivity information, for example from elements in the mesh to degrees of freedom. A similar pattern occurs in molecular dynamics simulations and graph processing, so both the theory and the tools that we will develop in this chapter are generalizable to these domains.

Because of the irregular memory access pattern, our approach to loop optimization must be based on dynamic analysis, particularly on *inspector/executor schemes*. Our hypothesis, backed by the studies reviewed in

Section 2.3.2, is that dynamic loop optimization through inspector/executor schemes can improve the performance of the applications in which we are interested. Among the various dynamic loop optimizations, we target *sparse tiling*. In Section 2.3.3 we explained that sparse tiling aims to exploit data reuse across consecutive loops. We recall that this optimization can be seen as the composition of three transformations: loop fusion, loop tiling, and automatic parallelization.

To summarize, the three main issues that we attack in this chapter are as follows:

- Previous approaches to sparse tiling are all based upon "ad-hoc" inspector/executor strategies; that is, developed "by hand", per application. We seek for a generalized technique, applicable to arbitrary computations on unstructured meshes.
- For this research to be successful, we believe we need a fully automated system capable of optimizing real-world applications. Automation is essential because computational scientists must abstract from low level optimization. We aim for a mixed compiler/library based approach, to be integrated with a framework for solving PDEs.
- There are only very few studies tackling loop fusion and inter-node parallelization. We are aware of none in presence of irregular memory accesses. We describe a technique that can co-exist with distributed-memory parallelism. This is essential because most scientific simulations run on at least hundreds of nodes.

3.2 Tackling Real-World Applications

3.2.1 Mesh-based Numerical Methods for PDEs

Loop fusion and loop tiling are widely studied in the literature. In spite of a notable research effort, however, it is not clear how widespread these optimizations are in real-world applications. Most studies centre their experimentation on relatively simple benchmarks and single-node performance; this unfortunately does not expose the complexity and the limitations of most scientific codes. On the other hand, it has repeatedly been shown that applying tiling to "simple" memory-bound loop nests can result in considerable speed-ups. The most striking examples are stencil codes arising in the finite difference method **?**, BLAS routines such as matrix multiplication **?**, and image processing kernels **?**. Since numerical methods for partial differential equations (PDEs) are often structured as sequences of parallelizable "sweeps" over the discretized equation domain, often implemented as memory-intensive loop nests, the following questions arise naturally:

Applicability Can we adopt sparse tiling (i.e., loop fusion composed with loop tiling), whose application has traditionally been limited to simple loop nests, in real-life numerical methods for solving PDEs?

Lack of evidence Why, despite decades of research, is it so difficult to find successful examples of integration of loop tiling with production code?

Challenges What are the theoretical and technical challenges that we have to overcome to automate this optimization, such that an entire community of computational scientists can benefit from it?

In this chapter, we will try answering these questions. Our study focuses on a particular class of applications:

Irregular codes Unstructured meshes are often used to discretize the computational domain, since they allow an accurate representation of complex geometries. For this type of meshes, the connectivity is usually stored by means of adjacency lists (or any equivalent structure), which results in indirect memory accesses (e.g., `A[B[i]]`) within loop nests. Indirections break static analysis, thus making any compiler-based approach (e.g., polyhedral optimization) unsuitable for our context. Indirections also make data dependence analysis more difficult, since this must now take place at runtime, introducing additional overhead.

Realistic dataset Most complex simulations operate on at least gigabytes of data, requiring multi-node execution. Any tiling-based optimization we will consider must therefore cope well with MPI execution.

Automation, but no legacy code We view tiling as an "extreme optimization"; that is, as an optimization that should be tried at the end of the development process, once the experimental results have been validated, to improve the execution time. In our experience, however, it is extremely rare that computational scientists have the expertise to add any low level optimizations, such as loop tiling, to their codes. In addition, applying these optimizations at the source level makes the code impenetrable and, therefore, almost impossible to maintain and to extend. Our aim is a fully automated technique abstracted to the users through a simple "switch" (i.e., loop tiling on/off) and a tiny set of parameters to drive performance tuning (e.g., the tile size). Automation clearly comes at the price of implementation complexity; we will integrate our tiling system with a generic framework for mesh iteration. We are therefore not interested in supporting legacy code, in which key computational aspects (e.g., mesh iteration, MPI communication) are usually hidden, for modularity, underneath several layers of software.

To support this class of applications and in order to maximize the chances that our work can be used in a variety of simulations in the years to come, our approach builds on two pillars:

- a library for writing inspector/executor schemes (**?**) for tiling arbitrary computations on unstructured meshes (or graphs);
- a multilayer framework based on DSLs and runtime code generation that uses such a library to automate the applciation of loop tiling.

3.2.2 Hypotheses

The "bare" structure of a scientific code solving a PDE by applying a numerical method on a discretized domain is shown in Listing 6. In the next section, we will use this example to motivate two fundamental hypotheses:

Hypothesis 1 A wide range of loop nests in mesh-based codes is, from a computational viewpoint, memory-bound, and increasing the data reuse across consecutive loop nests may relieve this issue.

LISTING 6: ....

```
// Time-stepping loop (T = total number of iterations)
for t = 0 to T {
  // 1st sweep over the C cells of the mesh
  for i = 0 to C {
    buffer_0 = gather_data ( A[f(map[i])], ...  )
    ...
    kernel_1( buffer_0, buffer_1, ...  );
    scatter_data ( buffer_0, f(map[i]) )
  }
  Calc ( ...  );
  MPI_Comm ( ...);
  // 2nd sweep over the N nodes of the mesh
  for i = 0 to N {
    ...  // Similar to sweep 1
  }
  // Boundary conditions:  sweep over the BE boundary edges
  for i = 0 to BE {
    ...  // Similar to sweep 1
  }
}
```

Hypothesis 2 Automating loop tiling in this kind of codes presents challenges that none of the state-of-the-art technologies can overcome. Our approach aims to solve this problem.

3.2.3 Automating Loop Tiling is More Difficult than Commonly Thought

We identify three classes of problems that are either neglected or treated disjointly in the relevant literature.

Theoretical questions We first of all wonder whether loop tiling really is the optimization we are looking for.

Computational Boundedness Since most computational methods are structured as sequences of loop nests and each loop nest is characterized by its own operational intensity, the computational boundedness is a concept that does not refer to the whole application. Rather, some loop nests may be memory-bound, whereas others CPU-bound, due to the complexity of the equations being solved or the employment of high order function spaces. Clearly, if all loop nests in an application are CPU-bound, the benefits of tiling on data locality will tend to be marginal. Before applying an aggressive optimization such as loop tilling, it is therefore fundamental to study the computational behaviour of an application for: (i) determining what fraction of the execution time is due to memory-bound loop nests; (ii) understanding if CPU-boundedness can be eliminated by applying other optimizations (e.g., vectorization), or if it is instead a natural consequence of the kernels and the architecture.

Tiling vs Space Filling Curves It is our impression that different communities have disjointedly researched solutions to the problem of relieving the memory pressure of mesh-based computations for years. We view both tiling and space filling curves (SFCs) – and likewise sequential execution – as instances of possible scheduling functions for a given loop nest (see Section 2.3.1). Nevertheless, we could not find studies comparing the performance of the two approaches, denoting a lack of communication between people in two distinct yet potentially very close domains: compiler optimization and computational science.

Practical issues Loop tiling for structured meshes is widely covered in the literature. Several studies have tackled automation (e.g., polyhedral compilers), time-loop tiling (i.e., time skewing), exotic tile shapes for minimizing communication (e.g., diamond tiling), and so on. However, the following challenges tend to be neglected, or at least only marginally tackled.

Unstructured meshes A technique to tile arbitrary computations on unstructured meshes was missing until this thesis[1]. Some inspector-executor strategies for tiling specific benchmarks, as discussed in Section 3.3, were available though.

Time tiling and MPI We reiterate the fact that real computations almost always run on distributed memory machines. The multi-node parallelization is often carried out resorting to MPI. This is evident in Listing 6 where MPI calls are placed between two consecutive mesh sweeps, since the first cells loop produces data needed by the subsequent nodes loop. Distributed memory parallelism poses a big challenge for time tiling, because now tiles at the partition boundary need special handling.

---

[1] We reinforce once more that the generalized tiling algorithm is actually the result of a joint collaboration between .... See **?**.

Time tiling and extra code The `Comp(...)` function in Listing 6 denotes the possibility that additional computation is performed between consecutive sweeps. This could be, for instance, checkpointing or I/O. Also, conditional execution of loops (e.g., through `if-then-else`) breaks time tiling.

Legacy code is usually impenetrable Our experience is that real-life code for scientific simulation tends to hide the tiling opportunities that a polyhedral compiler, for example, would search for. As explained in **?**, common problems are: 1) finding tilable loop nests, which are usually hidden for code modularity; 2) handling of boundary conditions; 3) dist

Limitations inherent in the numerical method Two loops cannot be fused if they are separated by a global synchronization point. This is often a global reduction, either explicit (e.g., the first loop updates a global variable that will be needed by the second loop) or implicit (i.e., within an external function invoked in between the two loops, like in many iterative solvers for linear systems). By limiting the applicability of many loop optimizations, global synchronization points pose great challenges and research questions. If strong scaling is the primary goal and memory-boundedness is the key limiting factor, then fundamental questions are: (i) can the numerical method be reformulated to relieve the constraints on low level optimization (which requires a joint effort between people in different domains); (ii) can the tools be made more sophisticated to work around these problems; (iii) most importantly, will the effort be rewarded by significant performance improvements?

In this chapter, we will formulate answers to some of these problems.

3.3 Related Work

Loop Chain

The data dependence analysis that we will develop is based on an abstraction called *loop chain*, which was originally presented in **?**. This abstraction is sufficiently general to capture data dependency in programs structured as arbitrary sequences of loops. We will detail our loop chain abstraction in Section 3.4.

Inspector/Executor and Sparse Tiling

The loop chain is the mechanism by means of which we will generalize inspector/executor schemes for unstructured codes. Inspector/executor strategies were first formalized by **?** and used to fuse and tile loops for improving data locality and providing parallelism in **????**.

Sparse tiling, which we introduced in Section 2.3.2, is the most notable technique based upon inspection/execution. The term was coined by Strout et al. **??** in the context of the Gauss-Seidel algorithm and in **?** in the context of the moldyn benchmark. However, the technique was initially proposed by Douglas et al. **?** to parallelize computations over unstructured meshes, taking the name of *unstructured cache blocking*. The mesh was initially partitioned; the partitioning represented the tiling in the first sweep over the mesh. Tiles would then shrink by one layer of vertices for each iteration of the loop. This shrinking represented what parts of the mesh could be update in later iterations of the outer loop without communicating with the processes executing other tiles. The unstructured cache blocking technique also needed to execute a serial clean-up tile at the end of the computation. Mark Adams **?** also developed an algorithm very similar to sparse tiling, to parallelize Gauss-Seidel computations. The main difference between **??** and **?** was that in the former work the tiles fully covered the iteration space, so a sequential clean-up phase at the end could be avoided.

We reiterate the fact that all these approaches were either specific to individual benchmarks or general but not scheduling across loops (i.e., loop fusion). Filling this gap is one of the contributions of this chapter.

Automated Code Generation and Optimization for Mesh-Based Computations

The automated code generation technique presented in **?** examines the data affinity among loops and partitions the execution with the goal of minimizing communication between processes, while maintaining load balancing. This technique supports unstructured mesh applications (being based on an inspector/executor strategy) and targets distributed memory systems, although it does not exploit the loop chain abstraction and abstracts from loop optimization.

Automated code generation techniques, such as those based on polyhedral compilers (reviewed in Section 2.3.1), have been applied to structured mesh benchmarks or proxy applications. Notable examples are Bondhugula et al. [2008], **?**], **?**. There has been very little effort in providing evidence that these tools can be effective in real-world applications. Time-loop diamond tiling was applied in **??** to a proxy code, but experimentation is limited to a single-node.

Overlapped Tiling

In structured codes, Using multiple layers of halo, or "ghost", elements is a common optimization in structured codes **?**. Overlapped tilling (see Section 2.3.3) exploits the same principle to reduce communication at the expense of performing redundant computation along the boundary **?**. Several studies centered on overlapped tiling within single regular loop nests (mostly stencil-based computations) **???**. Techniques known as "communication avoiding" **??** also fall in this broader category. To the best of our knowledge, overlapped tiling for unstructured grids by automated code generation has been studied only analytically in Giles et al. [2012].

3.4 Generalized Inspector/Executor Schemes

In Section 3.1 we introduced that our approach to sparse tiling is based upon an *inspector/executor scheme automatically generated at execution time*. In this section, we explain what is required to to construct such an inspector/executor scheme. The automation of this process will be tackled in Section 3.7.

3.4.1 The Loop Chain Abstraction

The mechanism we build is based upon the *loop chain*, an abstraction originally introduced in **?**. Informally, a loop chain is a sequence of loops with no global synchronization points, with attached some extra information to enable run-time data dependence analysis.

We recall from Section 3.2 that the complexity – particularly the presence of indirect memory accesses – and the modularity of real-world unstructured mesh applications prevent most common static optimizations for data locality. The data dependence information carried by a loop chain is supposed to be used to replace compile-time with run-time optimization. The basic idea is that the user provides the loop chain; then the compiler modifies the source code adding two extra pieces of code, the inspector and the executor; finally, at run-time, the inspector will exploit the data dependence information to build a "scheduling", which will eventually be used by the executor.

Before diving into the description of the loop chain abstraction, we observe two notable facts:

- The run-time execution of the inspection may introduce a significant overhead. In many scientific computations, however, the data dependence information is static; or, in more informal words, "the mesh does not change over time". Usually, the loop chain will be located within the time loop of the application, so the inspection cost is amortized. If the mesh changes over time (e.g., in case of adaptive mesh refinement), the inspection needs be re-executed. We have devoted a significant effort in optimizing the inspection algorithm, with the hope that its cost tends to be negligible even in the extreme case of frequent changes in the data dependence pattern. Further details will be provided in the later sections.

- The loop chain is expected to be provided by the user. We reinforce the idea that, in our approach, we will have two possibilities for constructing loop chains: either explicitly, with users required to change their program by adding calls to an external library to build the inspector/executor scheme, or implicitly, with the loop chain built automatically "behind the scenes" through a higher level framework.

In **??**, a loop chain L is defined as follows:

- $\mathbb{L}$ defines a scope consisting of n loops, $L_0, L_1, ..., L_{n-1}$. There are no global synchronization points between the loops.
- $\mathbb{D}$ is a set of disjoint m data spaces, $D_0, D_1, ..., D_{m-1}$. Each loop accesses (reads from, writes to) a certain subset of these data spaces. An access can be either direct (e.g., `A[i]`) or indirect (e.g., `A[map(i)]`).
- $R_{L_l \to D_d}(\vec{i})$ and $W_{L_l \to D_d}(\vec{i})$, where the R and W access relations are defined over for each data space $D_d \in D$ and indicate which data locations in data space D_d an iteration $i \in L_l$ reads from and writes to respectively (e.g., if we have `A[map(i)] = ...` in loop L_j, the access relation $map_{L_j \to A}(\vec{i})$ will be available).

3.4.2 Loop Chains for Unstructured Mesh Computations

Because of our focus on unstructured mesh computations, and inspired by the programming and execution models of OP2 (see Section 2.2.3), we refine the loop chain definition as follows:

- $\mathbb{L}$ defines a scope consisting of n, $L_0, L_1, ..., L_{n-1}$. There are no global synchronization points between the loops.
- The execution order of a loop iterations does not influence the result. This means that given L_i, we can choose an arbitrary scheduling of iterations because there will not be loop-carried dependencies.
- $\mathbb{S}$ is a set of disjoint m sets, $S_0, S_1, ..., S_{m-1}$. Possible sets are the elements in the mesh or the degrees of freedom associated to a certain function. A loop iterates over a set, or iteration space. A dataset is logically associated with one set (i.e., each value in the dataset is associated with a set element).
- A set is logically split into three contiguous regions: core, boundary, and non-exec.
- M is a set of k maps, $M_0, M_1, ..., M_{k-1}$. A map of arity a is a vector-valued function $M : S_i \to S_j^0 \times S_j^1 \times ... \times S_j^{a-1}$. It connects each element of S_i to one or more elements in S_j. For example, if a

triangular cell c is connected to three vertices v_0, v_1, v_2, we have $M(c) = [v_0, v_1, v_2]$.

- A loop L_i over the iteration space S is associated with d descriptors, $D_0, D_1, ..., D_{d-1}$. Each descriptor D_j is a 2-tuple $D_j = <M, \text{ mode}>$; M is either a map from S_j to some other sets or the special placeholder $\perp$, which indicates that the access is direct to some data associated with S_j, whereas mode is one of $[r,\ w,\ i]$, respectively read, write and increment.

The striking difference with respect to the original definition is the lack of data spaces. Essentially, we abstract data spaces to the level of sets. Each set S_j is usually associated multiple data spaces, and loops often access different data spaces associated with S_j. The idea, which will become more clear in the next section, is to limit data dependence tracking at the sets level. This may conservatively return some "false positives", but it significantly improves the inspection cost since typically $|S| << |D|$.

LISTING 7: Section of a toy program that is used as a running example to illustrate the loop chain abstraction and show how the tiling algorithm works.

```
for t = 0 to T {
  // Loop over edges
  for e = 0 to E {
    x = X[e];
    tmp_0 = tmp + edges2vertices[e + 0];
    tmp_1 = tmp + edges2vertices[e + 1];
    kernel1 (x, tmp_0, tmp_1);
  }

  // Loop over cells
  for c = 0 to C {
    res = R[C];
    tmp_0 = tmp + cells2vertices[c + 0];
    tmp_1 = tmp + cells2vertices[c + 1];
    tmp_2 = tmp + cells2vertices[c + 2];
    kernel2 (res, tmp_0, tmp_1, tmp_2);
  }

  // Loop over edges
  for e = 0 to E {
    tmp_0 = tmp + edges2vertices[e + 0];
    tmp_1 = tmp + edges2vertices[e + 1];
    kernel3 (tmp_0, tmp_1);
  }
}
```

Listing 7 shows a plain C implementation of the unstructured mesh OP2 program illustrated in Listing 5. A loop chain for this program is

LISTING 8: Building the loop chain for inspection.

```
nspector = init_inspector (...);

// Three sets, edges, cells, and vertices
E = set (inspector, nedges, ...);
C = set (inspector, ncells, ...);
V = set (inspector, nvertices, ...);

// Two maps:  from edges to vertices and from cells to vertices
e2vMap = map (inspector, E, V, edges2vertices, ...);
c2vMap = map (inspector, C, V, cells2vertices, ...);

// The loop chain comprises three loops; each loop has a set of descriptors
loop (inspector, E, {⊥, READ}, {e2vMap, INC});
loop (inspector, C, {⊥, READ}, {c2vMap, INC});
loop (inspector, E, {e2vMap, INC});

// Now can run the inspector
inspection = run_inspection (inspector, tile_size, ...)

```

constructed in Listing 8. The inspection output is stored in the data structure `inspection`. This is used (Listing **??**) to execute a "tiled" version of the original program. In the next section, we show through examples how the inspector exploits the information captured by the loop chain to construct tiles.

3.5 Sparse Tiling Examples

Using the example in Listing 7, we describe the actions performed by a sparse tiling inspector. We show two variants of inspection: for shared-memory and distributed-memory parallelism.

3.5.1 Overview of the Algorithm

The inspector starts with partitioning the iteration space of a *seed loop*, for example L_0. Partitions are used to initialize tiles: the edges in partition P_i – or, equivalently, the iterations of L_0 falling in P_i – are assigned to tile T_i. In the later stages of the inspection, as detailed in the upcoming sections, T_i will be populated with iterations from L_1 and L_2. The executor will assign a single thread/process to each T_i. T_i is executed "atomically"; that is, without the need for communication with other tiles. When executing T_i, first all iterations from L_0 are executed, then all iterations from L_1 and finally those from L_2. The challenge in scheduling iterations from L_j to T_i, therefore, is to guarantee that all data dependencies – read after write, write after read, write after write – are preserved.

3.5.2 Shared-Memory Parallelism

Figure **??** displays the situation after the initial partitioning of L_0.

There are four partitions, two of which are not connected through any edge or cell. These four partitions correspond to four tiles, $[T_0, T_1, T_2, T_3]$. Similarly to OP2, for shared-memory parallelism we adopt the following approach: (i) we serialize the execution of the elements inside a partition by scheduling them to a single thread; (ii) we color the partitions such that partitions assigned the same color can be executed in parallel by different threads. Two partitions can have the same color if they are not connected, because this ensures the absence of race conditions through indirect memory accesses. In the example we can use three colors: red (R), green (G), and blue (B). The two partitions at the top right and bottom left are not connected, so they are assigned the same color (green). In the following, with the notation T_i^c we mean that the i-th tile has color c.

In order to populate the tiles with iterations from L_1 and L_2, we first have to establish a total ordering for the execution of partitions with different colors. Here, we assume the following order: green (G), blue (B), and red (R). This means, for instance, that *all iterations* assigned to T_1^B must be executed *before all iterations* assigned to T_2^R. By "all iterations" we mean the iterations from L_0, as established by the seed partitioning, as well as the iterations that will later be assigned from L_1 and L_2. We assign integer positive numbers to colors to reflect their ordering, where a smaller number means higher execution priority. In this case, we can assign 0 to green, 1 to blue, and 2 to red.

To schedule the iterations of L_1 to $[T_0^G, T_1^B, T_2^R, T_3^G]$, we first need to compute a *projection* of any write or local reduction performed within L_0. A projection for L_0 is a function $\phi : V \to \mathbb{T}$ mapping a vertex (indirectly) incremented during the execution of L_0 to a tile. Consider the vertex v_0 in Figure **??**. v_0 has 7 incident edges, 2 of which belong to T_0^G, while the remaining 5 to T_1^B. Since we established that $G \prec B$, v_0 can only be read after T_1^B has finished executing the iterations from L_0 (i.e., the 5 incident blue edges). We express this condition by setting $\phi(v_0) = T_1^B$. Observe that we can compute ϕ by iterating over V and, for each vertex, applying the maximum function (MAX) to the color of the adjacent edges.

We now use ϕ to schedule L_1, a loop over cells, to the tiles. Consider again v_0 and the adjacent cells $[c_0,\ c_1,\ c_2]$. These three cells have in common the fact that they are adjacent to both green and blue vertices. For c_1, and similarly with the other cells, we compute $MAX(\phi(v_0),\ \phi(v_1),\ \phi(v_2)) = MAX(B, G, G) = B = 1$. This establishes that c_1 must be assigned to T_1^B, because otherwise (c_1 assigned instead to T_0^G) a read to v_0 would occur before the last increment from T_1^B took place. We indeed reiterate once more that the execution order is "all iterations from $[L_0, L_1, L_2]$ in the green tiles before all iterations from $[L_0, L_1, L_2]$ in the blue tiles".

In order to schedule L_2 to $[T_0^G,\ T_1^B,\ T_2^R,\ T_3^G]$ we employ a similar process. Vertices are again written by L_1, so a new projection over V will be computed and will be used in place of ϕ to schedule the edges in L_2. Figure **??** shows the result of this last phase.

Conflicting Colors

It is worth noting how T_2^R "consumed" the frontier elements of all other tiles every time a new loop was scheduled. Tiling a loop chain consisting of k loops had the effect of expanding the frontier of a tile of at most k vertices. With this in mind, we re-inspect the loop chain of the running example, although this time with a different execution order: blue (B), red (R), and green (G). We recall that the total ordering of colors is arbitrary, so this situation is legal. By applying the same inspection procedure that we just described, T_0^G and T_3^G will eventually become connected (see Figure **??**), thus violating the precondition "tiles/partitions with the same color can run in parallel". Indeed, race conditions during the execution of iterations belonging to L_2 are now theoretically possible. We will explain how to handle this problem in Section 3.6.

3.5.3 Distributed-Memory Parallelism

If parallelism is achieved through message-passing, the mesh and all datasets defined over it are distributed to different processes. Similarly to the example in Listing 6, MPI calls now separate the execution of two consecutive loops in the chain. This means that our inspector/executor scheme will have to take extra care of data dependencies along the mesh partition boundaries.

We logically split sets into three regions, *core*, *boundary*, and *non-exec*. The boundary region includes all iterations that cannot be executed until the MPI communications have successfully terminated. We pick L_0 as seed loop and we initialize the tiles as follows:

1. We take the core region of L_0 and partition it into tiles. Unless we aim for a mixed distributed/shared-memory parallelization scheme, there is no need to reassign the same color to unconnected tiles since we now have a single process executing sequentially all tiles (as opposed to the case of shared memory parallelism discussed in Section 3.5.2). We can assign colors increasingly: T_i is given color i, so we can uniquely identify a tile with its color. As long as tiles with contiguous ID are also physically contiguous in the mesh, and assuming that colors are executed in ascending order (i.e., T_0, $T_1, ...,$ as in the shared memory case), this assignment retains spatial locality when "jumping" from executing T_i to T_{i+1}.

2. We replicate over the boundary region of L_0 what we just did for core. Note that by neatly separating core and boundary, there cannot be tiles crossing the two regions. Further, all tiles within boundary have greater color than those in core, posing a constraint on the execution order: no boundary tiles can be executed until all core tiles have been scheduled.

3. We map the whole non-exec region of L_0 to a single special tile, T_{ne}. This tile has highest color and will actually never be executed. Its role will be explained next.

In Figure **??**, the same mesh of Figure **??** is distributed to two processes and the output of the initialization phase is displayed. The inspection then proceeds as in the case of shared memory parallelism. The application of the MAX function when scheduling iterations from L_1 and L_2 to tiles makes higher color tiles (i.e., those that will be scheduled later at execution time) "grow over" lower color ones. In particular, the whole boundary region "expands" over core, and so does T_{ne} over boundary (see Figure **??**).

The executor starts with triggering the all MPI communications required for the execution of the loop chain; it proceeds to scheduling all core tiles, thus overlapping communication with computation; it finishes, once all core tiles have been executed and the MPI communications have been accomplished, with scheduling the boundary tiles.

We make two key observations concerning the correctness and the efficiency of the strategy:

Correctness This inspector/executor scheme relies on redundant computation along the mesh partition boundary. The depth of the boundary region grows proportionally with the length of the loop chain. Roughly speaking, if the loop chain consists of n loops, then the boundary region needs $n-1$ extra layers of elements. In Figure **??**, the elements [...] belong to R_1, although they also need be executed by R_0 in order for [...] to be correctly computed when T_x executes iterations from L_2. This is a conceptually simple expedient, which however poses a big challenge on software engineering.

Efficiency The underlying hypothesis is that the increase in data locality achieved through sparse tiling will outweigh the overhead introduced by the redundant computation. This is based on the consideration that, in real applications, the core region tends to be way larger than the boundary one. In addition, not all iterations along the boundary region need be redundantly executed at every loop. For example, if we consider Figure **??**, we see that the strip of elements [...] is not relevant any more for the correct computation of [...]. The non-exec tile T_{ne}, which we recall is not scheduled by the executor, is deliberately given the highest color to grow over the boundary region: this will leave dirty values in some datasets until the next round of MPI communications, but will not hurt correctness.

3.6 Formalization

3.6.1 Data Dependency Analysis for Loop Chains

As with all loop optimizations that reschedule the iterations in a sequence of loops, any sparse tiling must satisfy the data dependencies. The loop chain abstraction, which we have described in Section 3.4, provides enough information to construct an inspector which analyzes all of the dependencies in a computation and builds a legal sparse tiling. We recall that one of the main assumptions in a loop chain is that each loop is fully parallel or, equivalently, that there are no loop carried dependencies.

The descriptors in the loop chain abstraction enable a general derivation of the storage-related dependencies between loops in a loop chain. The storage related dependencies between loops can be described as either flow (read after write), anti (write after read), or output (write after write) dependencies. In the following, assume that loop L_x, having iteration space S_x, always comes before loop L_y, having iteration space S_y, in the loop chain. Let us identify a descriptor of a loop L with $m^{mode}_{S_i \to S_j}$: this simply indicates that the loop L_i has iteration space S_i and uses a map m to write/read/increment elements (respectively, mode $\in \{w, r, i\}$) in the space S_j.

The flow dependencies can then be enumerated by considering pairs of points ($\vec{i}$ and $\vec{j}$) in the iteration spaces of the two loops L_x and L_y:

$$\{\vec{i} \to \vec{j} \mid \vec{i} \in S_x \wedge \vec{j} \in S_y \wedge m^{w}_{S_x \to S_z}(\vec{i}) \cap m^{r}_{S_y \to S_z}(\vec{j}) \neq \emptyset\}.$$

Anti and output dependencies are defined in a similar way. The anti dependencies for all pairs of loops L_x and L_y are:

$$\{\vec{i} \to \vec{j} \mid \vec{i} \in S_x \wedge \vec{j} \in S_y \wedge m^{r}_{S_x \to S_z}(\vec{i}) \cap m^{w}_{S_y \to S_z}(\vec{j}) \neq \emptyset\}.$$

While the output dependencies between loops L_x and L_y are:

$$\{\vec{i} \to \vec{j} \mid \vec{i} \in S_x \wedge \vec{j} \in S_y \wedge m^{w}_{S_x \to S_z}(\vec{i}) \cap m^{w}_{S_y \to S_z}(\vec{j}) \neq \emptyset\}.$$

In essence, there is a storage-related data dependence between two iterations from different loops (and therefore between the tiles they are placed in) when one of those iterations writes to a data element and the other iteration reads from or writes to the same data element.

There are local reductions, or "reduction dependencies" between two or more iterations of the same loop when those iterations "increment" the same location(s); that is, when they read, modify with a commutative and associative operator, and write to the same location(s). The reduction dependencies in L_x are:

$$\{\vec{i} \rightarrow \vec{j} \mid \vec{i} \in S_x \wedge \vec{j} \in S_x \wedge m^i_{S_x \rightarrow S_z}(\vec{i}) \cap m^i_{S_x \rightarrow S_z}(\vec{j}) \neq \emptyset\}.$$

The reduction dependencies between two iterations within the same loop indicates that those two iterations must be executed atomically with respect to each other.

Dependencies between non-adjacent loops are also possible. Our algorithm takes care of them by inspecting *projections*. In order to schedule L to tiles, the "most recent" projection for each space accessed by L must be available. For example, assume that the space S is read by L_k and is written by both L_i and L_j, with the loop chain being such that $L_i \prec ... \prec L_j \prec ... \prec L_k$. Let us denote by ϕ_S the projection for S. Each time that a loop accessing S in write mode is inspected for scheduling, ϕ_S must be updated. When eventually scheduling L_k, therefore, an up-to-date snapshot of ϕ_S (i.e., the one produced right after tiling L_j) will be used to compute a legal tiling.

3.6.2 The Generalized Sparse Tiling Algorithms

3.6.3 Inspection

The pseudo-code for the generalized sparse tiling inspector is showed in Algorithm 9. The output is a schedule that supports mixed distributed/shared memory parallelism. In the following, we describe the main steps of the algorithm.

ALGORITHM 9: The inspection algorithm

Input: The loop chain, $\mathbb{L} = [L_0,\ L_1,\ ...,\ L_{n-1}]$

Input: The tile size, *ts*

Output: A set of tiles $\mathbb{T}$, populated with iterations from $L_i \in \mathbb{L}$

The seed loop A constraint of the inspector is that only the first loop of the chain, L_0, can be chosen as seed loop, or L_{seed}. Any loop in the chain could theoretically be picked as seed, but this would make supporting distributed memory parallelism cumbersome because of the non-trivial coordination required between core and boundary tiles.

In the special case in which only shared memory parallelism is requested (i.e., no need to distinguish between core and boundary tiles any more), the seed loop can be chosen arbitrarily. If we however choose L_{seed} such that $L_0 \prec ... \prec L_{seed} \prec ...$, then a mechanism for constructing tiles in the reverse direction ("backwards"), from L_{seed} towards L_0, is needed. In **?**, we propose two "symmetric" algorithms to solve this problem, *forward tiling* and *backward tiling*, with the latter using the MIN function in place of MAX. For ease of exposition, and since in the fundamental case of distributed memory parallelism we always have $L_{seed} = L_0$, we here neglect this distinction[2].

Creation of tiles The algorithm starts with partitioning S^{core}_{seed}, the core region of S_{seed}, into $\{P_0, P_1, ..., P_{m-1}\}$, such that $P_i \cap P_j = \emptyset$ and $\cup_{i=0}^{m-1} P_i = S_{core}$. The partitioning is sequential: S_{seed} is "chunked" every ts iterations, with P_{m-1} that may be smaller than all other partitions. Similarly, we partition S^{bound}_{seed} into k subsets.

We then create $m + k + 1$ tiles, $\mathbb{T} = \{T_0, ..., T_{m+k}\}$, one for each partition and one extra tile for S^{nexec}_{seed}. Each tile T_i has four fields: a region (where possible values are core, boundary and non-exec); a color; for each loop L, a list of iterations T_i^L (initially empty, to be populated by the inspection algorithm); for each loop, a list of "local maps", whose role will become clear next. This structuring is illustrated in Figure **??**. At this point we have, for each $T_i \in \mathbb{T}$, $T_i^{L_{seed}} = P_i$.

Each tile needs be assigned a color, which represents a scheduling priority. If shared memory parallelism is requested, color the tiles such that adjacent tiles are not given the same color. Otherwise, assign increasing colors to tiles (i.e., T_i given color i). The boundary tiles are always given colors higher than that of core tiles; the non-exec tile has highest color.

---

[2] The algorithm that is implemented in the SLOPE library, however, is more general and supports backwards tiling as well for the case in which distributed memory parallelism is not required.

In essence, this is to provide the executor with sufficient information to schedule all core tiles first, then all boundary ones, and finally the non-exec tile. The adjacency relation is determined through the maps available through the descriptors of the loop chain.

Population of tiles Let ϕ be the data structure tracking projections for each space in the loop chain. Let $\sigma_{seed} : S_{seed} \rightarrow \mathbb{T}$ be the tiling function for L_{seed}, which maps iterations to tiles as determined at step (3). Then, for each loop L_i in $[L_1, L_2, ..., L_{n-1}]$:

- Using σ_{i-1} and the descriptors of L_{i-1} as input to Algorithm **??**, update ϕ. Besides updating ϕ, the algorithm returns a piece of information indicating whether two different tiles having the same color will become adjacent after tiling L_i. Let $C \in \mathbb{N}^{m \times m}$ be a matrix tracking conflicting colors, with $C[i, j] = 1$ indicating that T_i and T_j have the same color and will eventually become adjacent.
- Using ϕ, compute σ_i as described in Algorithm 1.
- Invert σ_i and assign iterations to tiles.

Detection of conflicts If C indicates the presence of at least one conflict, say between T_i and T_j, we add a "fake connection" between T_i and T_j and loop back to the coloring stage. T_i and T_j are now connected, so they will receive different colors.

Local maps Local maps are used during execution in place of the global maps provided by the loop chain, allowing sequential loops without the need for double indirections.

Algorithm 1 (Tiling)

1.

3.6.4 Execution

Algorithm 2 (Sparse Tiling Executor)

1.

3.6.5 Correctness of Inspection

...

3.6.6 Computational Complexity of Inspection

...

3.7 Automation

3.7.1 SLOPE: a Library for Tiling Irregular Computations

...

3.7.2 PyOP2: a Runtime Library for Mesh Iteration

...

3.7.3 Firedrake/DMPlex: the S-depth mechanism for MPI

...

3.8 Performance Evaluation

...

3.8.1 Benchmarks

- Sparse Jacobi
- Airfoil
- Wave Explicit

3.8.2 Seigen: an Elastic Wave Equation Solver for Seismological Problems

...

Chapter 4

On Optimality of Finite Element Integration

...

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: the low level optimization of the 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, 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.

We will also provide insights into the effects of more "traditional" compiler optimizations, such as loop unroll, loop interchange, loop fusion and vector promotion.

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

- A number of low level transformations for optimizing the performance of assembly kernels. Some of these transformations are directly inspired by the structure 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.

Listing 10 illustrates the effect of padding and data alignment on top of generalized code motion applied to the weighted Laplace operator presented in Listing 1.

LISTING 10: The assembly kernel for the weighted Laplace operator in Listing 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;
    #pragma vector aligned
    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 either generalized code motion or pre-evaluation, and problem constants is large. For example, applying code motion to the Burgers example 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 of this idea is given in Listing 11. 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 11: The assembly kernel for the weighted Laplace operator in Listing 1 after application of expression splitting (on top of generalized code motion). In this example, the split factor is 2.

```
void weighted_laplace(double A[3][3], double **coords, double w[3]) {
  // Omitting redundant code
  ...
  for (int j = 0; j<3; j++) {
    for (int k = 0; k<3; k++) {
      A[j][k] += (T_0[k]*T_0[j])*det*W[i]*f0;
    }
  }
  for (int j = 0; j<3; j++) {
    for (int k = 0; k<3; k++) {
      A[j][k] += (T_1[k]*T_1[j])*det*W[i]*f0;
    }
  }
}
...
```

Splitting an expression (henceforth *split*) has, however, several drawbacks. Firstly, it increases the number of accesses to `A` in proportion to the "split factor", which is the number of sub-expressions produced. Also, depending on how splitting is done, it can lead to redundant computation. For example, the number of times the product `det*W3[i]` is performed is proportional to the number of sub-expressions, as shown in the code snippet. Further, it increases loop overhead, for example through additional branch instructions. Finally, it might affect register locality: for instance, the same array could be accessed in different sub-expressions, requiring a proportional number of loads be performed; this is not the case of the running example, though. Nevertheless, the performance gain from improved register reuse can still be greater if suitable heuristics are used. Our approach consists of traversing the expression tree and recursively splitting it into multiple sub-expressions as long as the number of variables independent of the innermost loop exceeds a certain threshold. This is elaborated in the next sections, and validated against empirical search in Section 5.3.2.

5.2.3 Model-driven Vector-register Tiling

LISTING 12: The assembly kernel for the weighted Laplace operator in Listing 1 after application of vector-register tiling (on top of generalized code motion, padding, and data alignment). In this example, the unroll-and-jam factor is 1.

```
void weighted_laplace(double A[3][3], double **coords, double w[3]) {
  // Omitting redundant code
  ...
  // Padded buffer (note:  both rows and columns)
  double _A[4][4] ALIGN = {{0.0}};

  for (int i = 0; i<3; i++) {
    // Omitting redundant code
    // ...
    for (int j = 0; j<4; j += 4)
      for (int k = 0; k<4; k += 4) {
        // Sequence of LOAD and SET intrinsics
        // Compute _A[0][0], _A[1][1], _A[2][2], _A[3][3]
        // One _mm256_permute_pd per k-loop LOAD
        // Compute _A[0][1], _A[1][0], _A[2][3], _A[3][2]
        // One _mm256_permute2f128_pd per k-loop LOAD
        // ...
      }
    // Scalar remainder loop (not necessary in this example)
  }
  // Restore the storage layout
  for (int j = 0; j<4; j += 4) {
    for (int k = 0; k<4; k += 4) {
      __m256d 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 the loop trip counts and the variables accessed are such that the vector-register pressure is high. Since the kernel's working set is expected to fit 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 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 weighted Laplace assembly kernel using *op-vect* is shown in Listing 12.

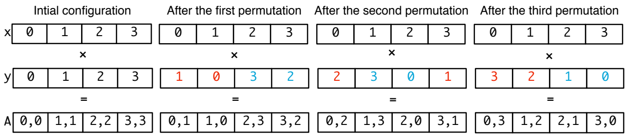
Intial configuration
After the first permutation
After the second permutation
After the third permutation
x
y
A
0 1 2 3
×
=
1 0 3 2
2 3 0 1
3 2 1 0
0,0 1,1 2,2 3,3
0,1 1,0 2,3 3,2
0,2 1,3 2,0 3,1
0,3 1,2 2,1 3,0

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

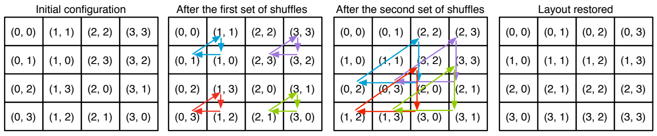
Initial configuration
After the first set of shuffles
After the second set of shuffles
Layout restored

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.

5.3 Experiments

5.3.1 Setup

The objective is to evaluate the impact of the code transformations presented in the previous sections in three representative PDEs, which we refer to as (i) Helmholtz, (ii) Diffusion, and (iii) Burgers.

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, 2014b]) 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 $q = 4$.

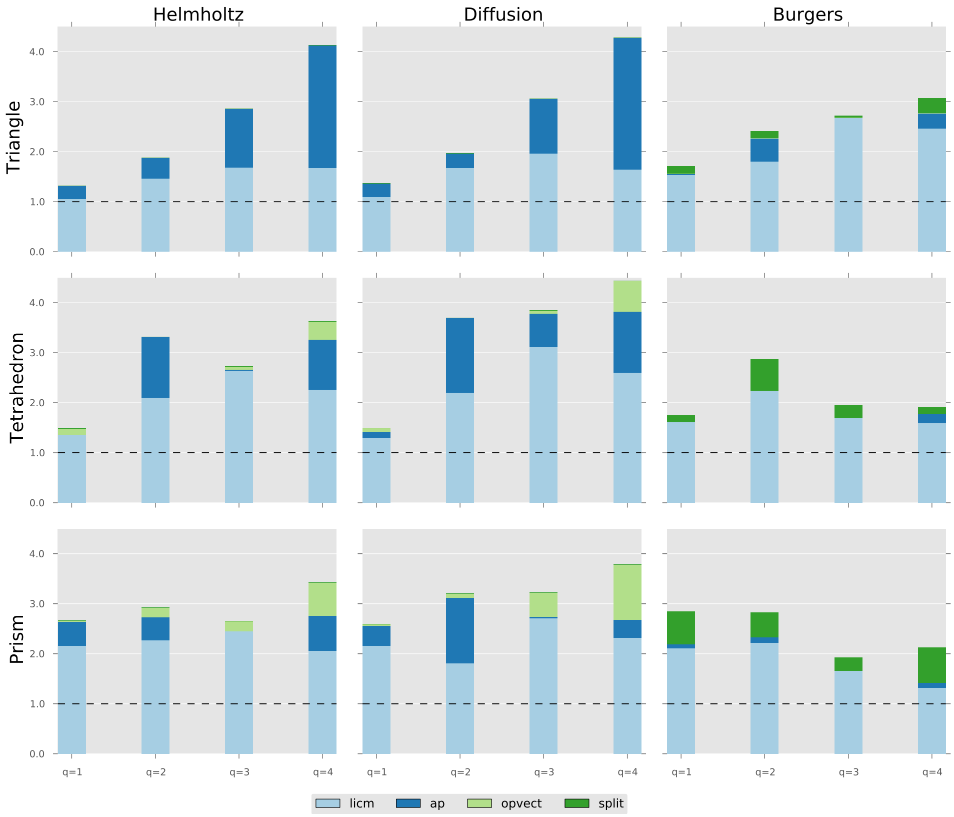

Figure 5.3: Performance improvement due to generalized loop-invariant code motion (*licm*), data alignment and padding (*ap*), outer-product vectorization (*op-vect*), and expression splitting (*split*) over the original non-optimized code. In each plot, the horizontal axis reports speed ups, whereas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

5.3.2 Impact of Transformations

Experiments were run on a single core of an Intel architecture, a Sandy Bridge I7-2600 CPU running at 3.4 GHz, 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 were tried, but they generally resulted in higher execution times).

The speed-ups achieved by applying the transformations on top of the original assembly kernel code are shown in Figure 5.3. This figure is a three-dimensional plot: element shape and equation vary along the outermost axes, whereas q varies within each sub-plot. In the next sections, we will refer to this figure and elaborate on the impact of the 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*.

Impact of Generalized Loop-invariant Code Motion

In general, the speed-ups 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 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 optimization 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 speed-ups 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 5.3.2, extensive unrolling already allows to achieve a certain degree of register re-use.

5.4 Experience with Traditional 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 the 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. 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.

Our experience with the Intel's and GNU's compilers is controversial: if, from one hand, the former applies this transformation following a reasonable cost model, the latter results in general more conservative, even at highest optimization level. This behaviour was verified in different variational forms (by looking at assembly code and compiler reports), including the complex hyperelastic model analyzed in Chapter 4.

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 10×10×10, 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 to implementations with 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. Our choice, therefore, is to leave the back-end compiler in charge of selecting unroll factors.

5.4.3 Vector promotion

Vector promotion is a transformation that "trades" space in exchange of a parallel dimension (a "clone" of the integration loop), thus promoting SIMD vectorization at the level of an outer loop.

LISTING 13: The assembly kernel for the weighted Laplace operator in Listing 1 after application of vector promotion (on top of generalized code motion).

```
void weighted_laplace(double A[3][3], double **coords, double w[3]) {
  // Omitting redundant code
  ...
  double f0[3] = {0.0};
  for (int i = 0; i<6; i++) {
    for (int r = 0; r < 3; ++r) {
      f0[i] += (w[r] * C[i][r]);
    }
  }
  for (int i = 0; i<6; i++) {
    double T_0[3] ALIGN;
    double T_1[3] ALIGN;
    for (int k = 0; k<3; 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++) {
      for (int k = 0; k<3; k++) {
        A[j][k] += (T_0[k]*T_0[j] + T_1[k]*T_1[j])*det*W[i]*f0[i]);
      }
    }
  }
}
```

Consider Listing 13. The evaluation of the coefficient `w` at each quadrature point can be vectorized by "promoting" `f` from a scalar to a vector of size 3. Any other sub-expression hoisted at the level of the integration loop (as described in Chapter 4) can be transformed in a similar way. The impact of this optimization obviously increases with the number of operations involving coefficients. At the same time, the allocation of extra memory may lead to the same issues described in Section **??**. Loop tiling could be used to counteract this negative effect, although this would significantly increase the implementation complexity.

We have not seen this transformation being applied by neither the GNU's nor the Intel's compilers. In our experience – and in absence of loop tiling – the impact on execution time is difficult to predict. This transformation requires further investigation. Despite being fully implemented in COFFEE, it is therefore not applied in the default optimization process.

5.4.4 Loop Fusion

Loop fusion is a well-known compiler transformation that consists of merging a sequence of loops into a single one. This optimization can be applied by most general-purpose compilers. What we cannot expect these compilers to do, however, is identifying common sub-expressions across the fused loops – an optimization of domain-specific nature.

In assembly kernels arising from bilinear forms, test and trial functions may belong to the same function space. More interestingly, the same operators could be applied to both sets of functions. This would result in both linear loops having the same iteration space and common sub-expressions arising across them. To avoid this kind of redundant computation and simultaneously enforcing fusion, we implemented in COFFEE a specialized version of loop fusion. In our experiments, this optimization always resulted in relatively small performance improvements, ranging between 2% and 8%. Therefore, it is automatically enabled in the default optimization process.

5.5 Related Work

The code transformations presented are inspired by standard compilers optimizations and exploit several 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 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 integration. In this section, we discuss their applicability in other computational domains and, in general, their integrability within a general-purpose compiler.

There are neither conceptual nor technical reasons which prevent our transformations from being used in other (general-purpose, research, ...) compilers. It is challenging, however, to assess the potential of the presented optimizations is another computational domains, and to what extent they would be helpful for improving the full application performance. To answer these questions, we first need to go back to the origins of our study. The starting point of our work was the mathematical formulation of a finite element operator, expressible as follows

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

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 cer-

tain architectures (for example GPUs) this could be prohibitive due to memory requirements. Intra-kernel parallelization is one possible solution: a domain-specific compiler 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 our context, expression splitting has successfully been 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. In the context of automated code generation for finite element local assembly, COFFEE is the first compiler capable of introducing low-level optimizations to simultaneously maximize 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. 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. 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

Sharing elimination and pre-evaluation, which we presented in Chapter 4, as well as the low level optimizations discussed in Chapter 5, have been implemented in COFFEE[1], a mature, platform-agnostic compiler. COFFEE has fully been integrated with Firedrake, the framework based on the finite element method introduced in Section **??**. The code, which comprises more than 5000 lines of Python, 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 form compiler, the Two-Stage Form Compiler (TSFC) **?**, into one or more abstract syntax trees (ASTs) representing assembly kernels. ASTs are then passed to COFFEE for optimization. The output of COFFEE, C code, is eventually provided to PyOP2 [Markall et al., 2013], where just-in-time compilation and execution over the discretized domain take place. The flow and the compiler structure are outlined in Figure 6.1.

---

[1] COFFEE is the acronym for COmpiler For Fast Expression Evaluation.

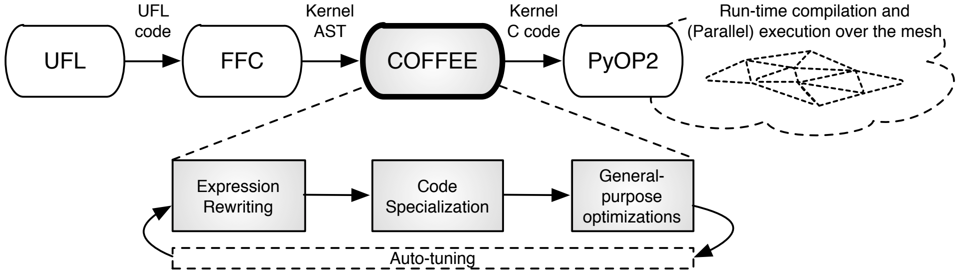
UFL
UFL code
FFC
Kernel AST
COFFEE
Kernel C code
PyOP2
Run-time compilation and (Parallel) execution over the mesh
Expression Rewriting
Code Specialization
General-purpose optimizations
Auto-tuning

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.

6.2 The Compilation Pipeline

Similarly to general-purpose compilers, COFFEE provides different optimization levels, namely `O0`, `O1`, `O2` and `O3`. Apart from `O0`, which does not transform the code received from the form compiler (useful for debugging purposes), all optimization levels apply ordered sequences of optimizations. In essence, the higher the optimization level, the more aggressive (and potentially slower) is the transformation process. In the following, when describing aspects of the optimization process common to `O1`, `O2` and `O3`, we will use the generic notation `Ox` ($\mathtt{x} \in \{1, 2, 3\}$).

The optimization level `Ox` can logically be split into three phases:

Expression rewriting Any transformation changing the structure of the expressions in the assembly kernel belongs to this class. For example, a high level optimization (sharing elimination, pre-evaluation) or, more in general, any rewrite operator (described later in Section 6.4) such as generalized code motion or factorization.

Handling of block-sparse tables Explained in Section **??**, this phase consists of restructuring the iteration spaces searching for a trade-off between the avoidance of useless operations involving blocks of zeros in basis function tables and the effectiveness of low level optimization.

Code Specialization The class of low level optimizations. The primary focus of this thesis has been code specialization for conventional

CPUs, although a generalization to other platforms is possible. In this phase, a specific combination of the transformations presented in Chapter 6 is applied.

These three phases are totally ordered. Expression rewriting introduces temporaries and creates loops. All loops, including those produced by expression rewriting, and the statements therein are potentially transformed in the subsequent phase, by adjusting bounds and introducing memory offsets, respectively. The output of the first two phases is finally processed for padding and data alignment, vector-register tiling and vector promotion.

Phase 1: analysis During the analysis phase, an AST is visited and several kinds of information are collected. In particular, COFFEE searches for expression rewriting candidates. These are represented by special nodes in the AST, which we refer to as "expression nodes". In plain C, we could think of an expression node as a statement preceded by a directive such as `#pragma coffee expression`; the purpose of the directive would be to trigger COFFEE's `Ox`. This is for example similar to the way loops are parallelized through OpenMP. If at least one expression node is found, we proceed to the next phase, otherwise the AST is unparsed and C code returned.

Phase 2: checking legality In addition to `Ox`, users can craft their own custom optimization pipelines by composing the individual transformations available in COFFEE. However, since some of the low level transformations are inherently not composable (e.g., loop unrolling with vector-register tiling), the compiler always checks the legality of the transformation sequence.

Phase 3: AST transformation If the sequence of optimizations is legal, the AST is processed. In particular:

`O1` At lowest optimization level, expression rewriting reduces to generalized code motion, while only padding and data alignment are applied among the low level optimizations.

02 With respect to `01`, there is only one yet fundamental change: expression rewriting now performs sharing elimination (i.e., Algorithm **??**).

03 Algorithm **??**, which coordinates sharing elimination and pre-evaluation, is applied. This is followed by handling block-sparse tables, and finally by padding and data alignment.

Phase 4: code generation Once all optimizations have been applied, the AST is visited one last time and a C representation (a string) is returned.

6.3 Plugging COFFEE into Firedrake

6.3.1 Abstract Syntax Trees

In this section, we highlight peculiarities of the hierarchy of AST nodes.

Special nodes Firstly, we observe that some nodes have special semantics. The expression nodes described in the previous section is one such example. A whole sub-hierarchy of `LinAlg` nodes is available, with objects such as `Invert` and `Determinant` representing basic linear algebra operation. Code generation for these objects can be specialized based upon the underlying architecture and the size of the involved tensors. For instance, a manually-optimized loop nest may be preferred to a BLAS function when the tensors are small[2]. Another special type of node is `ArrayInit`, used for static initialization of arrays. An `ArrayInit` wraps an N-dimensional Numpy array **?** and provides a simple interface to obtain information useful for optimization, like the sparsity pattern of the array.

Symbols A `Symbol` represents a variable in the code. The *rank* of a `Symbol` captures the dimensionality of a variable, with a rank equal to N indicating that a variable is an N-D array ($N = 0$ implies that the variable is a scalar). The rank is implemented as an N-tuple, each entry being either an integer or a string representing a loop dimension. The *offset* of a `Symbol` is again an N-tuple where each element is a 2-tuple. For each

---

[2] It is well-known that BLAS libraries are highly optimized for big tensors, while their performance tends to be sub-optimal with small tensors, which are very common in assembly kernels.

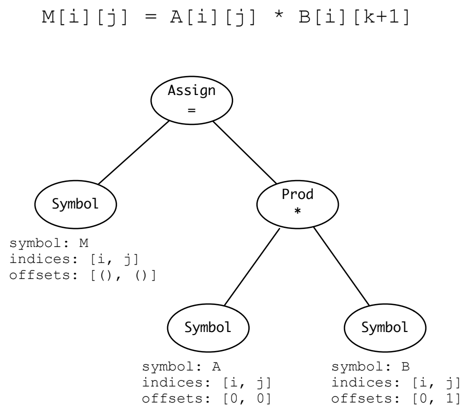

Figure 6.2: AST representation of a C assignment in COFFEE.

entry *r* in the rank, there is a corresponding entry <*scale*, *stride*> in the offset. Rank and offset are used as in Figure 6.2 to access specific memory locations. By clearly identifying rank and offset of a `Symbol` – rather than storing a generic expression – the complexity of the data dependency analysis required by the rewrite operators is greatly reduced. The underlying assumption, however, is that all symbols in the kernel (at least those relevant for optimization) have access functions (see Section 2.4.2) that are affine in the loop indices. As motivated in Chapter 4, this is definitely the case for the class of kernels in which we are interested.

Building an AST Rather than using a parser, COFFEE exposes to the user the whole hierarchy of nodes for explicitly building ASTs. This is because the compiler is meant to be used as an intermediate step in a multilayer framework based on DSLs. To ease the construction of ASTs (especially nested loops), a set of utility functions is provided. We will elaborate on these aspects in the next section.

6.3.2 Integration with Form Compilers

So far, COFFEE has been integrated with two form compilers: the FEniCS Form Compiler (FFC) and the Two-Stage Form Compiler (TSFC)[3]. These form compilers have their own internal representation of an assembly kernel; the objective is to turn such a representation into an AST suitable for COFFEE. We here describe how we achieved this in the case of FFC.

---

[3] The generation of ASTs in TSFC has been written by Myklos Homolya.

The key idea is to enrich the FFC's intermediate representation at construction time; that is, when the UFL specification of a form is translated. We made the following changes.

- The mathematical expression evaluating the element tensor is represented as a tree data structure, or "FFC-AST". A limitation of an FFC-AST was that its nodes – symbols or arithmetic operations – were not bound to loops. For instance, the FFC-AST node corresponding to the symbol `A[i][j]` did not separate the variable name `A` from the loop indices `i` and `j`. We have therefore enriched FFC-AST symbols with additional fields to capture these information.

- Basis functions in an FFC-AST are added a new field storing the dimensionality of their function space. This information is used to enrich `ArrayInit` objects with the sparsity pattern of the values they are representing (recall that the tabulation of vector-valued basis functions is characterized by the presence of zero-valued blocks).

The improved FFC-AST is intercepted prior to code generation (the last phase in the original FFC, which outputs C code directly) and forwarded to a new module, where a COFFEE AST is finally built. In this module:

- the template originally used by FFC for code generation (i.e., the parts of an assembly kernels that are immutable across different forms) is changed in favour of "static" pieces of AST (kernel signature, loop nests, etc).

- the FFC-AST is visited and translated into a COFFEE AST by a suitable AST-to-AST converter routine.

The Two-Stage Form Compiler was originally conceived to produce ASTs for COFFEE, so no particular changes to its intermediate representation were needed.

6.4 Rewrite Operators

COFFEE implements sharing elimination and pre-evaluation by composing "building-block" operators, or "rewrite operators". This has several advantages. Firstly, extendibility: novel transformations – for instance, sum-factorization in spectral methods – could be expressed using the existing operators, or with small effort building on what is already available. Secondly, generality: COFFEE can be seen as a lightweight, low level computer algebra system, not necessarily tied to finite element integration. Thirdly, robustness: the same operators are exploited, and therefore stressed, by different optimization pipelines. The rewrite operators, whose implementation is based on manipulation of the kernel's AST, essentially compose the COFFEE language.

The most important rewrite operators in COFFEE are:

Generalized code motion It pre-computes the values taken by a sub-expression along an invariant dimension. This is implemented by introducing a temporary array per invariant sub-expression and by adding a new "clone" loop to the nest (Several examples, e.g. Figure **??**, have been provided throughout the thesis). At the price of some extra memory for storing temporaries, all lifted terms are now amenable to auto-vectorization.

Expansion This transformation consists of expanding (i.e., distributing) a product between two generic sub-expressions. Expansion has several effects, the most important ones being exposing factorization opportunities and increasing the operation count. It can also help relieving the register pressure within a loop, by allowing further code motion.

Factorization Collecting, or factorizing, symbols reduces the number of multiplications and potentially exposes, as illustrated through sharing elimination, code motion opportunities.

Symbolic evaluation This operator evaluates sub-expressions that only involve statically initialized, read-only arrays (e.g., basis function tables). The result is stored into a new array, and the AST modified accordingly

All these operators are used by both sharing elimination and pre-evaluation (apart from symbolic evaluation, only employed by pre-evaluation).

The rewrite operators accept a number of options to drive the transformation process. With code motion, for example, we can specify what kind of sub-expressions should be hoisted (by indicating the expected invariant loops) or the amount of memory that is spendable in temporaries. Factorization can be either "explicit", by providing a list of symbols to be factorized or a loop dimension along which searching for factorizable symbols, or "heuristic", with the algorithm searching for the groups of most recurrent symbols.

6.5 Features of the Implementation

Rather than providing the pseudo-code and an explanation for each of the algorithms implemented in COFFEE – a mere exercise of scarce interest for the reader, given that the implementation is open-source and well-documented – this section focuses on the structure of the compiler and its "toolkit" for implementing or extending rewrite operators.

6.5.1 Tree Visitor Pattern

The need for a generic infrastructure for traversing ASTs has grown rapidly, together with the complexity of the compiler. In the early stages of COFFEE, any time that a new transformation (e.g., a rewrite operator) or data collector (e.g., for dependence analysis) were required, the full AST traversal had to be (re-)implemented. In addition, the lack of a common interface for tree traversals made the code more difficult to understand and to extend. This led to the introduction of a tree visitor design pattern[4], whose aim is to decouple the algorithms from the data structure on which they are applied **?**.

Consider, without loss of generality, an algorithm that needs to perform special actions (e.g., collecting loop dependence information) any time a `Symbol` or a `ForLoop` nodes are encountered. Then, a tree visitor will only need to implement three methods, namely `visit_Symbol` and

---

[4] The tree visitor infrastructure was mainly developed by Lawrence Mitchell, and was inspired by that adopted in UFL, the language used to specify forms in Firedrake.

`visit_ForLoop` – the actual handlers – as well as `visit_Node`, which implements the "fallback" action for all other node types (typically, just a propagation of the visit).

Tree visitors exploit the hierarchy of AST nodes by always dispatching to the most specialized handler. For example, symbols are simultaneously of type `Symbol` and `Expression`, but if a `Symbol` is encountered and `visit_Symbol` is implemented, then `visit_Symbol` is executed, whereas `visit_Expression` (if any) is ignored.

Most of the algorithms in COFFEE exploit the tree visitor pattern; a few, the "oldest" ones, still do not, due to the lack of time for porting to the new infrastructure.

6.5.2 Flexible Code Motion

Code motion consists of lifting, or hoisting, a (sub-)expression out of one or more loops. This rewrite operator is used in many different contexts: as a stand-alone transformation (optimization level `O1`); in multiple steps during sharing elimination; in pre-evaluation.

When applying the operator, several pieces of information must be known:

1. What sub-expression should be hoisted; for instance, should they be constant in the whole loop nest or invariant in at most one of the linear loops.
2. Where to hoist it; that is, how many loops is the operator allowed to cross.
3. How much memory are we allowed to use for a temporary.
4. If a common sub-expression had already been hoisted.

The code motion operator is flexible and let the caller (i.e., a higher-level transformation) drive the hoisting process by specifying how to behave with respect to the aforementioned points.

COFFEE must therefore track all of the hoisted sub-expressions for later retrieval. A dictionary mapping each of the temporaries introduced to a tuple of metadata is employed. For a temporary `t`, the dictionary records:

- A reference to the hoisted expression `e` assigned to `t`.

- A reference to the loop in which `e` is lifted.
- A reference to the declaration of `t`.

This dictionary belongs to the "global state" of COFFEE. It is updated each time the code motion operator is invoked, and read by other transformations (e.g., by all of the lower level optimizations).

The code motion operator "silently" applies common sub-expression elimination. A look-up in the dictionary tells whether a hoistable sub-expression `e` has been assigned to a temporary `t` by a prior call to the operator; in such a case, `e` is straightforwardly replaced with `t`, that is, no further temporaries are introduced.

6.5.3 Tracking Data Dependency

Data dependency analysis is necessary to ensure the legality of some transformations. For example:

- When lifting a sub-expression `e`, we may want to hoist "as far as possible" in the loop nest (possibly even outside of it); that is, right after the last write to a variable read in `e`.
- When expanding a product, some terms may be aggregated with previously hoisted sub-expressions. This would avoid introducing extra temporaries and increasing the register pressure. For example, if we have `(a + b)*c` and both `a` and `b` are temporaries created by code motion, we could expand the product and aggregate `c` with the sub-expressions stored by `a` and `b`. Obviously, this is as long as neither `a` nor `b` are accessed in other sub-expressions.
- For loop fusion (see Section 5.4.4).

In a similar way to general-purpose compilers, COFFEE uses a dependency graph for tracking data dependencies. The dependency graph has as many vertices as symbols in the code; a direct edge from `A` to `B` indicates that symbol `B` depends on (i.e., is going to read) symbol `A`. Since COFFEE relies on *static single assignment* – a property that ensures that variables are assigned exactly once – such a minimalistic data structure suffices for data dependence analysis.

6.5.4 Minimizing Temporaries

Both code motion operator (Section 6.5.2) and common sub-expression elimination induced by loop fusion (Section 5.4.4) impact the number of temporaries in the assembly kernel. At the end of expression rewriting, a routine in COFFEE attempts to remove all of the unnecessary temporaries. This makes the code more readable and, potentially, relieves the register pressure.

The main rule for removing a temporary `t` storing an expression `e` is that if `t` is accessed only in a single statement `s`, then `e` is inlined into `s` and `t` is removed. Secondly, if some of the transformations in the optimization pipeline reduced `e` to a symbol, then any appearance of `t` is also replaced by `e`.

Chapter 7

Conclusions

...

Bibliography

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`.

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`.

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`.

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`.

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, C. Bertolli, P. H. J. Kelly, E. Laszlo, and I. Reguly. An analytical study of loop tiling for a large-scale unstructured mesh application. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 477–482, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4956-9. doi: 10.1109/SC.Companion.2012.68. URL `http://dx.doi.org/10.1109/SC.Companion.2012.68`.

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/.

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 individual transformations in COFFEE. `https://github.com/firedrakeproject/firedrake/tree/pyop2-ir-perf-eval/tests/perf-eval`, 2014b.

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.

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.

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`.

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`.

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`.

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`.