

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

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Thesis Statement . . . . .	1
1.2	Overview . . . . .	1
1.3	Contributions . . . . .	2
1.4	Dissemination . . . . .	2
1.5	Thesis Outline . . . . .	3
<b>2</b>	<b>Background</b>	<b>5</b>
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
<b>3</b>	<b>Automated Loop Fusion and Tiling for Irregular Computations</b>	<b>15</b>
3.1	Motivation . . . . .	15
3.2	Tackling Real-World Unstructured Mesh Applications . . .	16
3.2.1	Open Problems, Questions, Hypotheses . . . . .	16
3.2.2	Applying Loop Fusion and Tiling is Much More Dif- ficult than Commonly Thought . . . . .	18
3.3	Related Work . . . . .	21
3.4	Generalized Inspector/Executor Schemes . . . . .	23
3.4.1	Relationship between Loop Chain and Inspector . .	23
3.4.2	The Loop Chain Abstraction . . . . .	24
3.4.3	The Loop Chain Abstraction Revisited for Unstruc- tured Mesh Computations . . . . .	25
3.5	Loop Chain and Inspection Examples . . . . .	27
3.6	Formalization . . . . .	34
3.6.1	Data Dependency Analysis for Loop Chains . . . . .	34
3.6.2	The Generalized Sparse Tiling Inspector . . . . .	35
3.6.3	The Generalized Sparse Tiling Executor . . . . .	40
3.6.4	Computational Complexity of Inspection . . . . .	42
3.7	Implementation . . . . .	42
3.7.1	SLOPE: a Library for Tiling Irregular Computations	44
3.7.2	PyOP2: a Runtime Library for Mesh Iteration based on Lazy Evaluation . . . . .	45
3.7.3	Firedrake/DMPlex: the S-depth mechanism for MPI	46
3.8	Performance Evaluation . . . . .	47
3.8.1	Benchmarks . . . . .	47
3.8.2	Seigen: an Elastic Wave Equation Solver for Seismo- logical Problems . . . . .	51
3.9	Conclusions and Future Work . . . . .	52
<b>4</b>	<b>On Optimality of Finite Element Integration</b>	<b>53</b>

<b>5</b>	<b>Cross-loop Optimization of Arithmetic Intensity for Finite Element Integration</b>	<b>55</b>
5.1	Recapitulation and Objectives . . . . .	55
5.2	Low-level Optimization . . . . .	58
5.2.1	Padding and Data Alignment . . . . .	58
5.2.2	Expression Splitting . . . . .	59
5.2.3	Model-driven Vector-register Tiling . . . . .	61
5.3	Experiments . . . . .	63
5.3.1	Setup . . . . .	63
5.3.2	Impact of Transformations . . . . .	65
5.4	Experience with Traditional Compiler Optimizations . . . .	69
5.4.1	Loop Interchange . . . . .	69
5.4.2	Loop Unroll . . . . .	69
5.4.3	Vector promotion . . . . .	70
5.4.4	Loop Fusion . . . . .	71
5.5	Related Work . . . . .	71
5.6	Applicability to Other Domains . . . . .	72
5.7	Conclusion . . . . .	76
<b>6</b>	<b>COFFEE: a Compiler for Fast Expression Evaluation</b>	<b>77</b>
6.1	Overview . . . . .	77
6.2	The Compilation Pipeline . . . . .	78
6.3	Plugging COFFEE into Firedrake . . . . .	80
6.3.1	Abstract Syntax Trees . . . . .	80
6.3.2	Integration with Form Compilers . . . . .	81
6.4	Rewrite Operators . . . . .	83
6.5	Features of the Implementation . . . . .	84
6.5.1	Tree Visitor Pattern . . . . .	84
6.5.2	Flexible Code Motion . . . . .	85
6.5.3	Tracking Data Dependency . . . . .	86
6.5.4	Minimizing Temporaries . . . . .	87
<b>7</b>	<b>Conclusions</b>	<b>89</b>





# List of Tables

2.1	Type and variable names used in the various listings to identify local assembly objects. . . . .	8
3.1	Summary of the notation used throughout the section. . . .	36
3.2	The tile data structure. . . . .	38
3.3	Worst-case costs of inspection. . . . .	42
3.4	Execution time reductions over the original implementation (in percentage) and speed-ups over the single-threaded tiled implementation for the sparse Jacobi solver with 15 threads. . . . .	48
3.5	Execution time (in seconds) and speed-ups over the slowest single-threaded implementation for the Airfoil benchmark. The values are obtained from simulations with fully-loaded machines (16 and 24 threads/processes on the Sandy Bridge and the Westmere architectures, respectively). . . . .	49



# List of Figures

2.1	Structure of a local assembly kernel . . . . .	7
3.1	Partitioning of the seed loop. . . . .	28
3.2	Tiling $L_0$ . . . . .	29
3.3	Projection of $L_0$ to $L_1$ . . . . .	30
3.4	Tiling $L_1$ . . . . .	30
3.5	Tiling $L_2$ . . . . .	31
3.6	Tiling the program in Listing 7 for shared memory parallelism can lead to conflicts. Here, the two green tiles eventually become adjacent, exposing potential race conditions.	32
3.7	Seed partitioning for distributed memory execution. . . . .	33
5.1	Outer-product vectorization by permuting values in a vector register. . . . .	63
5.2	Restoring the storage layout after <i>op-vect</i> . The figure shows how $4 \times 4$ 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. . . . .	63
5.3	Performance improvement due to generalized loop-invariant code motion ( <i>licm</i> ), data alignment and padding ( <i>ap</i> ), outer-product vectorization ( <i>op-vect</i> ), and expression splitting ( <i>split</i> ) 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. . . . .	65

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. . . . .	78
6.2	AST representation of a C assignment in COFFEE. . . . .	81

# 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 Finit 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.

---

```

1 void weighted_laplace(double A[3][3], double **coords, double w[3])
2 {
3     // Compute Jacobian
4     double J[4];
5     compute_jacobian_triangle_2d(J, coords);
6
7     // Compute Jacobian inverse and determinant
8     double K[4];
9     double detJ;
10    compute_jacobian_inverse_triangle_2d(K, detJ, J);
11    const double det = fabs(detJ);
12
13    // Quadrature weights
14    static const double W[6] = 0.5;
15
16    // Basis functions
17    static const double B[6][3] = {{...}} ;
18    static const double C[6][3] = {{...}} ;
19    static const double D[6][3] = {{...}} ;
20
21    for (int i = 0; i < 6; ++i) {
22        double f0 = 0.0;
23        for (int r = 0; r < 3; ++r) {
24            f0 += (w[r] * C[i][r]);
25        }
26        for (int j = 0; j < 3; ++j) {
27            for (int k = 0; k < 3; ++k) {
28                A[j][k] += (((((K[1]*B[i][k])+(K[3]*D[i][k])) *
29                    ((K[1]*B[i][j])+(K[3]*D[i][j])))) +
30                    ((K[0]*B[i][k])+(K[2]*D[i][k])) *
31                    ((K[0]*B[i][j])+(K[2]*D[i][j])))))*det*W[i]*f0;
32            }
33        }
34    }
35 }

```

---

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 ( $\phi$ ) 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  $\alpha_3$  is represented by the  $r$  loop. In this example, we assume  $d = 2$  (2D mesh), so the summations along  $\alpha_1$ ,  $\alpha_2$  and  $\beta$  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.

---

```

1 // This is a Firedrake construct (not an UFL's) to instantiate a 2D mesh.
2 mesh = UnitSquareMesh(size, size)
3 // FunctionSpace also belongs to the Firedrake language
4 V = FunctionSpace(mesh, "Lagrange", 2)
5 u = TrialFunction(V)
6 v = TestFunction(V)
7 weight = Function(V).assign(value)
8 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 research 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 specification 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  $a = \text{weight} * \dots$  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 suffices 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.

---

```

1 void helmholtz(double M[3][3], double **coords) {
2   // K, det = Compute Jacobian (coords)
3
4   static const double W[3] = {...}
5   static const double A[3][3] = {...}
6   static const double B[3][3] = {...}
7
8   for (int i = 0; i < 3; i++)
9     for (int j = 0; j < 3; j++)
10      for (int k = 0; k < 3; k++)
11        M[j][k] += (Y[i][k]*Y[i][j]+
12                    +((K1*A[i][k]+K3*B[i][k])*(K1*A[i][j]+K3*B[i][j]))+
13                    +((K0*A[i][k]+K2*B[i][k])*(K0*A[i][j]+K2*B[i][j])))*
14                *det*W[i];
15 }
```

---

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.

---

```

1 void burgers(double A[12][12], double **coords, double **w) {
2   // K, det = Compute Jacobian (coords)
3
4   static const double W[5] = {...}
5   static const double A[5][12] = {...}
6   static const double B[5][12] = {...}
7   //11 other basis functions definitions.
8   ...
9   for (int i = 0; i<5; i++) {
10    double f0 = 0.0;
11    //10 other declarations (f1, f2,...)
12    ...
13    for (int r = 0; r<12; r++) {
14      f0 += (w[r][0]*C[i][r]);
15      //10 analogous statements (f1, f2, ...)
16      ...
17    }
18    for (int j = 0; j<12; j++)
19      for (int k = 0; k<12; k++)
20        A[j][k] += (.(K5*F9)+(K8*F10))*Y1[i][j])+
21          +(((K0*C[i][k])+(K3*D[i][k])+(K6*A[i][k]))*Y2[i][j]))*f11)+
22          +(((K2*E[i][k])+...+(K8*B[i][k]))*((K2*E[i][j])+...+(K8*B[i][j]))))+
23          + <roughly a hundred sum/muls go here>..)*
24          *det*W[i];
25    }
26  }

```

---

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: ...

```
1 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))
2 // 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))
3 // 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 Loop Fusion and Tiling for Irregular Computations

### 3.1 Motivation

Many numerical methods for partial differential equations (PDEs) are structured as sequences of parallel loops. This exposes parallelism well, but does not convert data reuse between loops into data locality, since large datasets are usually accessed. In Section 2.3.1 we have explained that loop fusion and loop tiling could be used to retain some of this potential data locality. As we elaborate in the upcoming sections, however, it is extremely challenging to implement these optimizations in most real-world applications.

Our focus is on unstructured mesh PDE solvers, like 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 transformation is based on dynamic analysis, particularly on *inspector/ex-*

*ecutor schemes*. Our hypothesis, backed by the studies reviewed in Section 2.3.2, is that dynamic loop optimization can improve the performance of a class of real-world unstructured mesh applications. Among the possible dynamic loop optimizations, we target *sparse tiling*. We recall from Section 2.3.3 that sparse tiling aims to exploit data reuse across consecutive loops by composing three transformations: loop fusion, loop tiling, and automatic parallelization.

The three main issues that we tackle in this chapter are:

- Previous approaches to sparse tiling were 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.
- Automation is more than a desired feature because application specialists neglect, or simply ignore, complicated optimizations that harm source code comprehensibility. We aim for a fully-automated framework based upon a mixed compiler/library approach.
- Very few studies have tackled loop fusion in presence of inter-node parallelization. We are aware of none for the case in which the memory accesses pattern is irregular, as in unstructured mesh applications. We describe a technique for the co-existence of loop fusion and distributed-memory parallelism. This is a fundamental contribution because most scientific simulations run on multi-node architectures.

## 3.2 Tackling Real-World Unstructured Mesh Applications

### 3.2.1 Open Problems, Questions, Hypotheses

Loop fusion and loop tiling have been widely studied in the literature. In spite of a notable research effort, however, it is still 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 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. Among the most notable examples there are stencil codes arising in finite difference methods ?, 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 loops, or “sweeps”, over the discretized equation domain, the following questions arise naturally:

**Applicability** Can we adopt sparse tiling (i.e., fusion and tiling) in real-life numerical methods for solving PDEs? If so, what kind of gain in performance should we expect?

**Lack of evidence** Why, despite decades of research, loop transformations are rarely used in scientific (production) codes?

**Challenges** What are the theoretical and technical challenges that we have to overcome to automate sparse tiling, thus making it available to entire communities of computational scientists?

We tackle these questions in the following context:

**Irregular codes** Unstructured meshes are often used to discretize the computational domain, since they allow for an accurate representation of complex geometries. Their connectivity is stored by means of adjacency lists (or equivalent data structure). This leads to indirect memory accesses (e.g.,  $A[B[i]]$ ) within loop nests. Indirections break static analysis, thus making compiler-based approaches to loop transformation (e.g., polyhedral optimization) unsuitable for our context. Runtime data dependence analysis enables dynamic loop optimization, although at the price of additional overhead.

**Realistic datasets** Complex simulations usually operate on at least gigabytes of data, requiring multi-node execution. Any loop transformation we consider will have to work in conjunction with distributed-memory parallelism.

**Automation, but no legacy code** Sparse tiling is an “extreme optimization” requiring a great deal of effort to be implemented, especially due to the need for a complete restructuring of the simulation core. Similarly to many low level transformations, it also makes the source

code impenetrable, increasing the maintenance and the extension costs. We therefore aim for a fully automated system based on DSLs, which abstracts sparse tiling through a simple interface (i.e., a simple construct that lets the users tell the compiler “apply sparse tiling to the following sequence of loops” ) and a tiny set of parameters for performance tuning (e.g., the tile size). We are not in supporting legacy code, where the key computational aspects (e.g., mesh iteration, distributed-memory parallelism) are usually hidden for software modularity, thus making automation almost impossible.

Based upon these requirements, we structured our approach on two pillars:

- a library for writing inspector/executor schemes (?) for applying sparse tiling to arbitrary computations on unstructured meshes;
- integration of the library with a multilayer framework based on DSLs and runtime code generation.

Finally, to summarize, our hypotheses are:

- Many loop nests in unstructured mesh applications are memory-bound, so exploiting the data reuse across consecutive loop nests should improve the performance.
- Applying sparse tiling is extremely complicated, so automation is necessary. However, automation presents challenges that none of the existing technologies can overcome.

### 3.2.2 Applying Loop Fusion and Tiling is Much More Difficult than Commonly Thought

We show in Listing 6 the “skeleton” of a typical PDE solver on an unstructured mesh. This will be useful throughout the analysis presented in this section.

We identify three classes of problems that are either completely neglected or treated with scarce emphasis in the relevant literature.

**Theoretical questions** First of all, we wonder whether sparse tiling really is the optimization we need in unstructured mesh applications.

---

**LISTING 6:** The “bare” structure of a numerical method for solving a partial differential equation. Three parallelizable sweeps over sets of mesh components – cells, nodes, boundary nodes – are executed within a time-stepping loop. In the cells loop, we show how a kernel is invoked: first, the memory indirections are resolved; the kernel is then invoked passing data which is now contiguous in memory (this hopefully increases the chances of vectorisation); finally, the result is “scattered” back to memory. Distributed-memory parallelism is achieved through MPI; the exchange of messages (`MPI.Comm (...)`) occurs in between different mesh sweeps. Additional computation (`Calc (...)`) could also be present (e.g., sparse linear algebra operations, as in the finite element method; checkpointing for fault tolerance).

---

```

1 // Time-stepping loop (T = total number of iterations)
2 for t = 0 to T {
3   // 1st sweep over the C cells of the mesh
4   for i = 0 to C {
5     buffer_0 = gather_data ( A[f(map[i])], ... )
6     ...
7     kernel_1( buffer_0, buffer_1, ... );
8     scatter_data ( buffer_0, f(map[i]) )
9   }
10  Calc (...);
11  MPI.Comm (...);
12  // 2nd sweep over the N nodes of the mesh
13  for i = 0 to N {
14    ... // Similar to sweep 1
15  }
16  // Boundary conditions: sweep over the BV boundary nodes
17  for i = 0 to BV {
18    ... // Similar to sweep 1
19  }
20  ...
21  Calc (...);
22  MPI.Comm (...);
23  ...
24 }
```

---

**Computational Boundedness** Computational methods for PDEs are structured as sequences of loop nests, each loop nest characterized by its own operational intensity. In the same application, some loop nests may be memory-bound, while others CPU-bound. This clearly depends on several parameters of the numerical method, including the arithmetic complexity of the operators and the discretization employed (e.g., polynomial order of function spaces). Obviously, if most loop nests in a code are CPU-bound, the benefits of sparse tiling on data locality will be marginal. Before even thinking about aggressive optimizations such as fusion and tiling, it is fundamental to understand

the limiting factors of an application. In essence, two questions need be answered: (i) what fraction of the execution time is due to memory-bound loop nests; (ii) can CPU-boundedness be relieved by applying other optimizations (e.g., vectorization).

**Tiling vs Space Filling Curves** Loop tiling and Space Filling Curves (SFCs) are two solutions produced by different communities to the same problem: improving the performance of mesh-based computations by making a better use of memory. Our view is that both tiling and SFCs are simply instances of scheduling functions for a given loop nest (see Section 2.3.1). We could not find studies comparing the performance of the two approaches. This seems to denote a lack of communication between communities that often tackle similar problems.

**Practical issues** Several studies on loop fusion and tiling for structured mesh applications have covered automation (e.g., polyhedral compilers), time-loop tiling (i.e., time skewing), exotic tile shapes for communication minimization (e.g., diamond tiling), and many other problems. However, the following problems are rarely given the attention they would deserve.

**Unstructured meshes** A technique to fuse and tile arbitrary computations on unstructured meshes was missing until this thesis<sup>1</sup>. Ad-hoc inspector-executor strategies for some proxy applications, as discussed in Section 3.3, were available though. Unstructured meshes are fundamental because they are used in a wide range of numerical methods.

**Time tiling and MPI** We reiterate the fact that real-world computations almost always run on distributed memory architectures, relying on MPI. This is evident in Listing 6 where MPI calls appear between consecutive mesh sweeps. Distributed-memory parallelism poses a big challenge for time tiling, because tiles at the partition boundary need special handling.

**Time tiling and extra code** The `Comp(...)` function in Listing 6 denotes the possibility that additional computation is performed

---

<sup>1</sup>We reinforce once more that the generalized sparse tiling algorithm is the result of a joint collaboration among the authors of ?.



between consecutive sweeps. This could be, for instance, check-pointing or I/O. Also, conditional execution of loops (e.g., through `if-then-else`) breaks time tiling.

**Legacy code is usually impenetrable** Loop transformation opportunities are often hidden in existing scientific codes. As explained in ?, common problems are: 1) potentially fusible or tiling loop nests are separated for code modularity; 2) handling of boundary conditions; 3) source code not amenable for data dependency analysis (e.g., extensive use of pointers, function calls).

**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 is read by the second loop) or implicit (i.e., within an external function invoked 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 interesting questions are: (i) can the numerical method be reformulated to relieve the constraints on low level optimization (which requires a joint effort between application and performance specialists); (ii) can the tools be made more sophisticated to work around these problems; (iii) will the effort be rewarded by significant performance improvements.

### 3.3 Related Work

#### Loop Chain

The data dependence analysis that we develop in this chapter 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 [1] and used to fuse and tile loops for improving data locality and providing parallelism in [2].

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. [3] in the context of the Gauss-Seidel algorithm and in [4] in the context of the moldyn benchmark. However, the technique was initially proposed by Douglas et al. [5] 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 [6] also developed an algorithm very similar to sparse tiling, to parallelize Gauss-Seidel computations. The main difference between [3] and [6] 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 [7] 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 tiling (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 this section we explain how to build inspector/executor schemes for arbitrary sequences of loops.

### 3.4.1 Relationship between Loop Chain and Inspector

The *loop chain* is an abstraction 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 indirect memory accesses inhibit static optimization for data locality in real-world unstructured mesh applications. The idea is to exploit the data dependence information carried by a loop chain to replace compile-time with run-time optimization. Users are supposed to provide the loop chain; a high level compiler then modifies the source code adding the inspector and replacing a sequence of loops with another piece of code, the executor; at run-time, the inspector

exploits the data dependence information in the loop chain and builds a “scheduling”, which is eventually used as input to the executor.

Before diving into the description of the loop chain abstraction, it is worth observing the following:

- The execution of the inspection phase introduces an overhead. In many scientific computations, however, the data dependence pattern is static; or, more informally, “the mesh does not change over time”. This means that the inspection cost can be amortized over multiple iterations of the time loop. If instead the mesh changes over time (e.g., in case of adaptive mesh refinement), the inspection needs be executed again. We have spent a considerable amount of time in designing and implementing a highly optimized inspector algorithm; this hopefully will make the overhead negligible even in the unfortunate case of frequent changes to the data dependence pattern.
- We have explained that the loop chain must be provided by the user. A user can be an application specialist, who manually changes a program by constructing the loop chain and the inspector/executor scheme through library calls, or even a higher layer of abstraction, in which case the optimization is fully automated. The tools we have built allow both approaches.

Further details on these two points will be provided in the later sections.

### 3.4.2 The Loop Chain Abstraction

In ??, a loop chain  $\mathbb{L}$  is defined as follows:

- $\mathbb{L}$  consists of  $n$  loops,  $L_0, L_1, \dots, L_{n-1}$ . There are no global synchronization points between the loops. The execution order of a loop’s 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.
- $\mathbb{D}$  is a set of disjoint  $m$  data spaces,  $D_0, D_1, \dots, D_{m-1}$ . 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.,  $A[\text{map}(i)]$ ).

- $R_{L_i \rightarrow D_d}(\vec{i})$  and  $W_{L_i \rightarrow D_d}(\vec{i})$ , where the  $R$  and  $W$  access relations are defined over for each data space  $D_d \in D$ . They indicate which data locations in data space  $D_d$  an iteration  $i \in L_i$  reads from and writes to respectively (e.g., if we have  $A[B(i)] = \dots$  in loop  $L_j$ , the access relation  $B_{L_j \rightarrow A}(\vec{i})$  will be available).

### 3.4.3 The Loop Chain Abstraction Revisited 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), the definition of a loop chain  $\mathbb{L}$  is refined as follows:

- $\mathbb{L}$  consists of  $n$  loops,  $L_0, L_1, \dots, L_{n-1}$ . There are no global synchronization points between the loops. The execution order of a loop's 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.
- $S$  is a set of disjoint  $m$  sets,  $S_0, S_1, \dots, S_{m-1}$ . 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 a set (i.e., each value in the dataset is associated with a set element).

A set  $S$  is logically split into three contiguous regions: core ( $S^c$ ), boundary ( $S^b$ ), and non-exec ( $S^{ne}$ ). Given a process  $P$  and a set  $S$ , we have that:

$S^c$  The iterations of  $S$  that exclusively belong to  $P$ .

$S^b$  The boundary region can be seen as the union of two sub-regions, owned ( $S^{\text{owned}}$ ) and exec ( $S^{\text{exec}}$ ).  $S^{\text{owned}}$  are iterations that belong to  $P$  which will be redundantly executed by some other processes;  $S^{\text{exec}}$  are iterations from other processes which are redundantly executed by  $P$ . We will see that redundant computation is exploited to preserve atomic execution of a tile – the property that enables execution without the need for synchronization.

$S^{ne}$  These are iterations of other processes that are communicated to  $P$  because they need be indirectly read to correctly compute  $S^b$ .

A set is uniquely identified by a name and the sizes of its three regions. For example, the notation  $S = (\text{vertices}, C, B, N)$  defines the `vertices` set, which comprises  $C$  elements in the core region (iterations 0 to  $C - 1$ ),  $B$  elements in the boundary region (iterations  $C$  to  $C + B - 1$ ), and  $N$  elements in the non-exec region (iterations  $C + B$  to  $C + B + N - 1$ ).

- The *depth* is an integer indicating the extent of the boundary region of a set. This constant is the same for all sets.
- $\mathbb{M}$  is a set of  $k$  maps,  $M_0, M_1, \dots, M_{k-1}$ . A map of arity  $a$  is a vector-valued function  $M : S_i \rightarrow S_j^0 \times S_j^1 \times \dots \times S_j^{a-1}$  that 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,  $D_0, D_1, \dots, D_{d-1}$ . Each descriptor  $D_j$  is a 2-tuple  $D_j = \langle M, \text{mode} \rangle$ .  $M$  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  $S_j$ . *mode* is one of  $[r, w, i]$ , meaning that a memory access is respectively of type read, write or increment.

With respect to the original definition, one of the most important differences is the lack of data spaces. In unstructured mesh applications, a loop tends to access multiple data spaces associated with the same set. The idea, therefore, is to rather rely on sets to perform data dependency analysis. This can significantly improve the inspection cost, because typically  $|S| \ll |\mathbb{D}|$ .

The second fundamental difference is the characterization of sets into the three regions core, boundary and non-exec. This separation is essential for distributed-memory parallelism (we simply have that both  $S^b$  and  $S^{ne}$  contain no elements in the special case of execution entirely based on shared-memory parallelism). The “extent” of the boundary regions is captured by the *depth* of the loop chain; the actual role of this parameter will become clear in Section 4.

---

**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.

---

```

1 for t = 0 to T {
2   // Loop over edges
3   for e = 0 to E {
4     x = X[e];
5     tmp_0 = tmp + edges2vertices[e + 0];
6     tmp_1 = tmp + edges2vertices[e + 1];
7     kernel1 (x, tmp_0, tmp_1);
8   }
9
10  // Loop over cells
11  for c = 0 to C {
12    res = R[C];
13    tmp_0 = tmp + cells2vertices[c + 0];
14    tmp_1 = tmp + cells2vertices[c + 1];
15    tmp_2 = tmp + cells2vertices[c + 2];
16    kernel2 (res, tmp_0, tmp_1, tmp_2);
17  }
18
19  // Loop over edges
20  for e = 0 to E {
21    tmp_0 = tmp + edges2vertices[e + 0];
22    tmp_1 = tmp + edges2vertices[e + 1];
23    kernel3 (tmp_0, tmp_1);
24  }
25 }

```

---

### 3.5 Loop Chain and Inspection Examples

Using the example in Listing 7 – a plain C implementation of the unstructured mesh OP2 program illustrated in Listing 5 – we describe the actions performed by a sparse tiling inspector. The inspector takes as input a loop chain, as illustrated in Listing 8. We will show two variants of inspection, for shared-memory and distributed-memory parallelism. Which variant is executed depends on the value of the variable `mode` at line 18 in Listing 8.

#### Overview

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, in other words, the iterations of  $L_0$  falling in  $P_i$  – are assigned to tile  $T_i$ . Figure 3.1 displays the situation after the initial partitioning of  $L_0$  for a given input mesh. There are four partitions, two of which are not connected through any edge or cell. These four partitions correspond to four tiles,  $[T_0, T_1, T_2, T_3]$ .

---

**LISTING 8:** Building the loop chain for inspection.

---

```
1 nspector = init_inspector (...);
2
3 // Three sets, edges, cells, and vertices
4 E = set (inspector, core_edges, boundary_edges, nonexec_edges, ...);
5 C = set (inspector, core_cells, boundary_cells, nonexec_cells, ...);
6 V = set (inspector, core_vertices, boundary_vertices, nonexec_vertices, ...);
7
8 // Two maps: from edges to vertices and from cells to vertices
9 e2vMap = map (inspector, E, V, edges2vertices, ...);
10 c2vMap = map (inspector, C, V, cells2vertices, ...);
11
12 // The loop chain comprises three loops; each loop has a set of descriptors
13 loop (inspector, E, {⊥, READ}, {e2vMap, INC});
14 loop (inspector, C, {⊥, READ}, {c2vMap, INC});
15 loop (inspector, E, {e2vMap, INC});
16
17 // Now can run the inspector
18 inspection = run_inspection (mode, inspector, tile_size, ...)
19 return inspection;
```

---

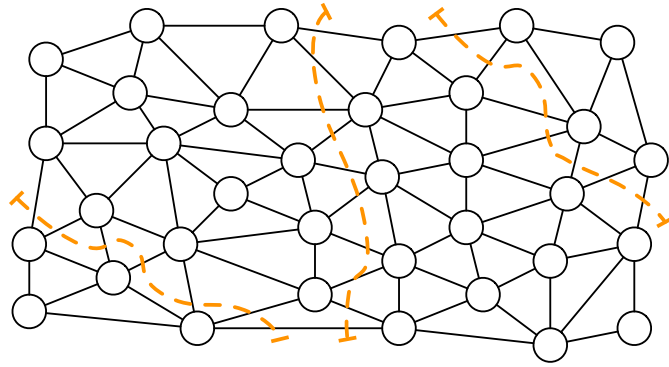


Figure 3.1: Partitioning of the seed loop.

In the later stages of the inspection, as detailed in the next sections,  $T_i$  is populated with iterations from  $L_1$  and  $L_2$ . The executor assigns a single thread/process to each  $T_i$ .  $T_i$  is executed “atomically”; that is, without ever communicating 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  $T_i$  is to guarantee that all data dependencies – read after write, write after read, write after write – are preserved.



## Shared-Memory Parallelism

Similarly to OP2, for shared-memory parallelism we apply tiles coloring. Two tiles that are given the same color can be executed in parallel by different threads. Two tiles 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).  $T_0$  and  $T_3$  are not connected, so they are assigned the same color. The colored tiles are shown in Figure 3.2. In the following, with the notation  $T_i^c$  we indicate that the  $i$ -th tile has color  $c$ .

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  $L_0$  (determined 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  $\phi : V \rightarrow \mathbb{T}$  mapping a vertex (indirectly) incremented during the execution of  $L_0$  to a tile. Consider the vertex  $v_0$  in Figure 3.3.  $v_0$  has 7 incident edges, 2 of which belong to  $T_0^G$ , while the

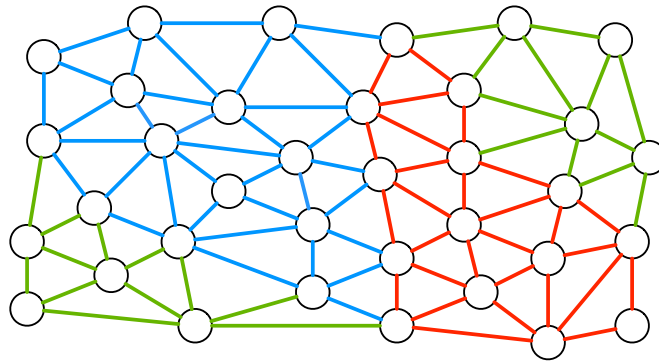


Figure 3.2: Tiling  $L_0$ .

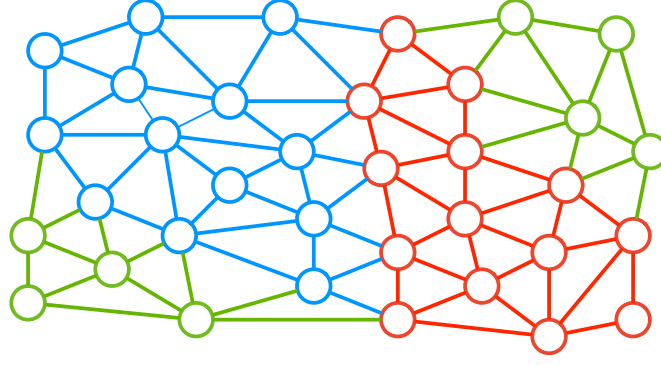


Figure 3.3: Projection of  $L_0$  to  $L_1$ .

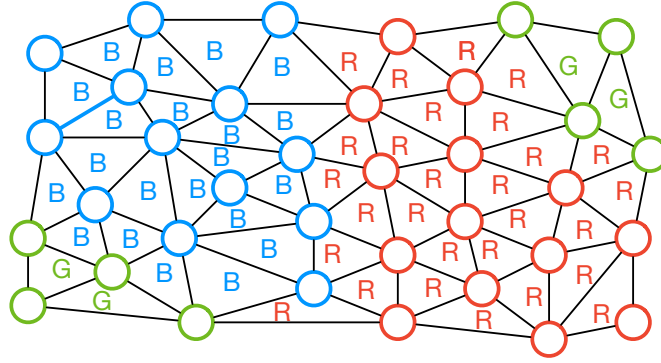


Figure 3.4: Tiling  $L_1$ .

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  $\phi$  by iterating over  $V$  and, for each vertex, applying the maximum function ( $MAX$ ) to the color of the adjacent edges.

We now use  $\phi$  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”. The result of

scheduling  $L_1$  to tiles is displayed in Figure 3.4.

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  $\phi$  to schedule the edges in  $L_2$ . Figure 3.5 shows the result of this last phase.

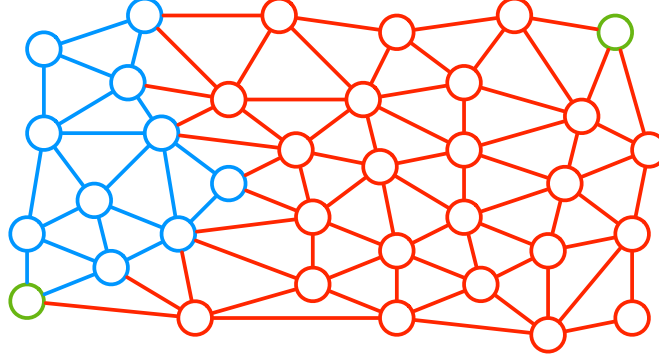


Figure 3.5: Tiling  $L_2$ .

### 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 3.6), 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 3.6.2.

### Distributed-Memory Parallelism

If parallelism is achieved through message-passing, the mesh and its datasets are distributed to different processes. Similarly to the example in Listing 6, MPI calls now separate the execution of two consecutive loops in

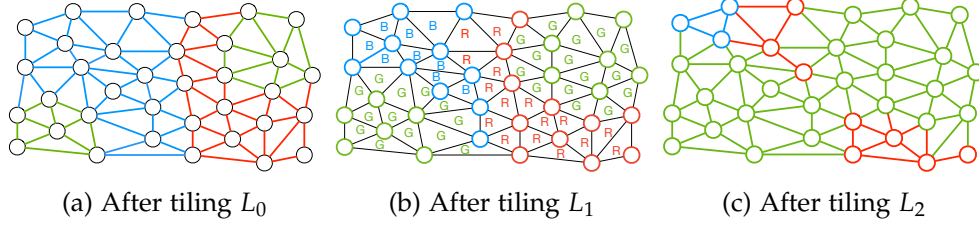


Figure 3.6: Tiling the program in Listing 7 for shared memory parallelism can lead to conflicts. Here, the two green tiles eventually become adjacent, exposing potential race conditions.

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 the 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). 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.,  $T_0, T_1, \dots$ , 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 3.7, the same mesh of Figure 3.1 has been 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 ??).

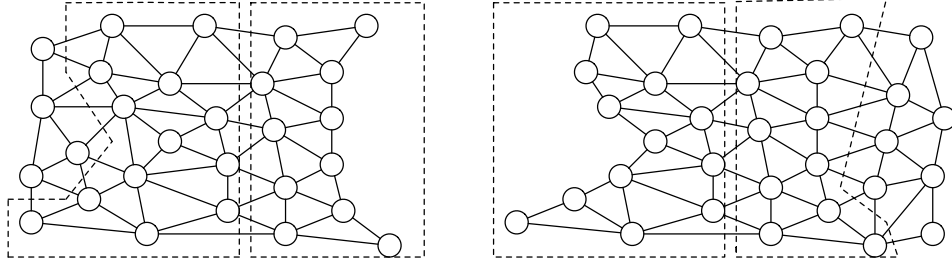


Figure 3.7: Seed partitioning for distributed memory execution.

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 finally observe that:

**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 3.7, 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 intro-

duced 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  $m_{S_i \rightarrow S_j}^{mode}$ : 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  $L_y$ :

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

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

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

While the output dependencies between loops  $L_x$  and  $L_y$  are:

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

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:

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

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

As seen in the example in Section 3.5, our inspector algorithm handles data dependencies, including those between non-adjacent loops, by tracking *projections*. In the next section we explain how projections are constructed and used.

### 3.6.2 The Generalized Sparse Tiling Inspector

The pseudo-code for the generalized sparse tiling inspector is showed in Algorithm 1. Given a loop chain and an average tile size, the algorithm produces a schedule suitable for mixed distributed/shared memory par-

Symbol	Meaning
$\mathbb{L}$	The loop chain
$L_i$	The $i$ -th loop in $\mathbb{L}$
$S_i$	The iteration space of $L_i$
$S_i^c, S_i^b, S_i^n$	the core, boundary, and non-exec regions of $S_i$
$S$	A generic set in $\mathbb{L}$
$S_{in}, S_{out}$	The input and output sets of a map
$D$	A descriptor of a loop
$r, w, i$	Possible values for $D.mode$
$\mathbb{T}$	The set of all tiles
$\mathbb{T}[i]$	Accessing the $i$ -th tile
$T_i^c, T_i^b$	the $i$ -th tile over the core and boundary regions
$\phi_S$	A projection $\phi_S : S \rightarrow \mathbb{T}$
$\Phi$	The set of all available projections
$\sigma_i$	The tiling function $\sigma_i : S_i \rightarrow \mathbb{T}$ for $L_i$

Table 3.1: Summary of the notation used throughout the section.

allelism. In the following, we elaborate on the main steps of the algorithm. The notation used throughout the section is summarized in Table 3.1.

**Choice of the seed loop** The seed loop  $L_{seed}$  is used to initialize the tiles. Theoretically, any loop in the chain can be chosen as seed. Supporting distributed memory parallelism, however, is cumbersome if  $L_{seed} \neq L_0$ . This is because more general partitioning and coloring schemes would be needed to ensure that no iterations in any  $S_i^b$  are assigned to a core tile. A constraint of our inspector algorithm in the case of distributed memory parallelism, therefore, is that  $L_{seed} = L_0$ .

In the special case in which there is no need to distinguish between core and boundary tiles (e.g., because a program is executed on a single shared memory system), the seed loop can be chosen arbitrarily. If we however pick  $L_{seed}$  in the middle of the loop chain ( $L_0 \prec \dots \prec L_{seed} \prec \dots$ ), a mechanism for constructing tiles in the reverse direction (“backwards”), from  $L_{seed}$  towards  $L_0$ , is necessary. 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* when computing projections. For ease of exposition, and since in the fundamental case of distributed memory parallelism we impose  $L_{seed} = L_0$ , we here neglect



---

**ALGORITHM 1:** The inspection algorithm

---

**Input:** The loop chain  $\mathbb{L} = [L_0, L_1, \dots, L_{n-1}]$ , a tile size  $ts$

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

---

```
// Initialization
1 seed  $\leftarrow 0$ ;
2  $\Phi \leftarrow \emptyset$ ;
3  $C \leftarrow \perp$ ;

// Creation of tiles
4  $\sigma_{seed}, \mathbb{T} \leftarrow \text{partition}(S_{seed}, ts)$ ;
5 seed_map  $\leftarrow \text{find\_map}(S_{seed}, \mathbb{L})$ ;
6 conflicts  $\leftarrow \text{false}$ ;
7 do
    // Schedule loops to tiles
    color( $\mathbb{T}$ , seed_map);

    for  $i = 1$  to  $n - 1$  do
        project( $L_{i-1}, \sigma_{i-1}, \Phi, C$ );
         $\sigma_i \leftarrow \text{tile}(L_i, \Phi)$ ;
        assign( $\sigma_i, \mathbb{T}$ );
    end for

    if found_conflicts( $C$ ) then
        conflicts  $\leftarrow \text{true}$ ;
        add_fake_connection(seed_map,  $C$ );
    end if
18 while not conflicts;

// Inspection successful, create local maps and return
19 create_local_maps( $\mathbb{T}$ );
20 return  $\mathbb{T}$ 
```

---

this distinction<sup>2</sup>.

**Tiles initialization** The algorithm starts with partitioning  $S_{seed}^c$  into  $m$  subsets  $\{P_0, P_1, \dots, P_{m-1}\}$  such that  $P_i \cap P_j = \emptyset$  and  $\cup_{i=0}^{m-1} P_i = S_{seed}^c$ . 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  $S_{seed}^b$  into  $k$  subsets.

We then create  $m + k + 1$  tiles,  $\mathbb{T} = \{T_0^c, \dots, T_{m-1}^c, T_m^b, \dots, T_{m+k-1}^b, T_{m+k}^n\}$ ,

---

<sup>2</sup>The algorithm that is implemented in the SLOPE library, however, is more general and supports backwards tiling for the case in which distributed memory parallelism is not required.

Field	Possible values
region	core, boundary, non-exec
color	an integer representing the execution priority
iterations lists	one list of iterations (integers) $T_i^{L_j}$ for each $L_j \in \mathbb{L}$
local maps	one list of local maps for each $L_j \in \mathbb{L}$ ; one local map for each map used in $L_j$

Table 3.2: The tile data structure.

one for each partition and one extra tile for  $S_{seed}^n$ .

A tile  $T_i$  has four fields, as summarized in Table 3.2. The region is used by the executor to schedule tiles in a certain order. The value of this field is determined right after the seed partitioning, since a tile is exclusively in one of  $\{S_{seed}^c, S_{seed}^b, S_{seed}^n\}$ . The iterations lists represent the iterations in  $\mathbb{L}$  belonging to  $T_i$ . For each  $L_j \in \mathbb{L}$ , there is one iterations list  $T_i^{L_j}$ . At the beginning, for each  $T_i \in \mathbb{T}$  we have  $T_i^{L_{seed}} = T_i^{L_0} = P_i$ , whereas all other lists are empty. Local maps are used by the executor in place of the global maps provided by the loop chain; we will see that this allows avoiding double indirections.

Each tile needs be assigned a color, which gives it a scheduling priority. If shared memory parallelism is requested, adjacent tiles are given different colors (the adjacency relation is determined through the maps available in  $\mathbb{L}$ ). Otherwise, the colors are assigned to the tiles in increasing order (i.e.,  $T_i$  is given color  $i$ ). The boundary tiles are always given colors higher than that of core tiles; the non-exec tile has highest color. In essence, this will allow the executor to schedule all core tiles first and all boundary tiles afterwards.

**Construction of tiles by tracking data dependencies** To schedule a loop to tiles we use projections. A projection is a function  $\phi_S : S \rightarrow \mathbb{T}$ . Projections are computed and/or updated after tiling a loop. Initially, the set  $\Phi$  of all projections is empty. Starting with the seed tiling  $\sigma_{seed} : S_{seed} \rightarrow \mathbb{T}$ , we iteratively update  $\Phi$  and build tiling functions for all other loops  $[L_1, L_2, \dots, L_{n-1}]$ .

**Projecting tiled sets** Algorithm 2 takes as input  $\sigma_{i-1}$  and (the descriptors of)  $L_{i-1}$  to update  $\Phi$ . Further, the conflicts matrix  $C \in \mathbb{N}^{m \times m}$ , indicating

---

**ALGORITHM 2:** Projection of a tiled loop

---

**Input:** A loop  $L_i$ , a tiling function  $\sigma_i$ , the projections set  $\Phi$ , the conflicts matrix  $C$   
**Result:** Update  $\Phi$  and  $C$

```
1 foreach  $D$  in  $L_i$ .descriptors do
2   if  $D.mode == r$  then
3      $\text{skip}$ ;
4   end if
5   if  $D.map == \perp$  then
6      $\Phi = \Phi \cup \sigma_i$ ;
7   else
8      $\text{inverse\_map} \leftarrow \text{map\_invert}(D.map)$ ;
9      $S_j, S_i, \text{values}, \text{offsets} \leftarrow \text{inverse\_map}$ ;
10     $\phi_{S_j} \leftarrow \perp$ ;
11    for  $e = 0$  to  $S_j.size$  do
12      for  $k = \text{offsets}[e]$  to  $\text{offsets}[e+1]$  do
13         $\text{adjacent\_tile} = \mathbb{T}[\text{values}[k]]$ ;
14         $\text{max\_color} \leftarrow \text{MAX}(\phi_{S_j}[e].color, \text{adjacent\_tile}.color)$ ;
15        if  $\text{max\_color} \neq \phi_{S_j}[e].color$  then
16           $\phi_{S_j}[e] \leftarrow \text{adjacent\_tile}$ ;
17        end if
18      end for
19    end for
20     $\text{update}(C, \mathbb{T}, \phi_{S_j})$ ;
21     $\Phi = \Phi \cup \phi_{S_j}$ ;
22  end if
23 end foreach
```

---

whether two different tiles having the same color will become adjacent after tiling  $L_{i-1}$ , is updated.

A projection tells what tile a set element logically belongs to at a given point of the tiling process. A new projection  $\phi_S$  is needed if the elements of  $S$  are written by  $L_i$ . Let us consider the non-trivial case in which writes or increments occur indirectly through a map  $M : S_i \rightarrow S_j^0 \times S_j^1 \times \dots \times S_j^{a-1}$ . To compute  $\phi_{S_j}$ , we first determine the inverse map (an example is shown in Figure ??). Then, we iterate over all elements of  $S_i$  that indirectly access elements in  $S_j$ . In particular, given  $e \in S_j$ , we determine the last tile that writes to  $e$ , say  $T_{last}$ , through the application of the MAX function to tile colors. We then simply set  $\phi_{S_j}[e] = T_{last}$ .

---

**ALGORITHM 3:** Building a tiling function

---

**Input:** A loop  $L_i$  (with iteration space  $S_i$ ), the projections set  $\Phi$

**Output:** The tiling function  $\sigma_i$

---

```
1  $\sigma_i \leftarrow \perp$ ;  
2 foreach  $D$  in  $L_i.descriptors$  do  
3   if  $D.map == \perp$  then  
4      $\sigma_i \leftarrow \Phi[S_i]$ ;  
5   else  
6      $arity \leftarrow D.map.arity$ ;  
7      $\phi_S \leftarrow \Phi[D.map.S_j]$ ;  
8     for  $e = 0$  to  $S_i.size$  do  
9        $\sigma_i[e] \leftarrow T_\perp$  ;  
10      for  $k = 0$  to  $arity$  do  
11         $adjacent\_tile \leftarrow \phi_S[D.map.values[e*arity + k]]$ ;  
12         $max\_color \leftarrow \text{MAX}(\sigma_i[e].color, adjacent\_tile.color)$ ;  
13        if  $max\_color \neq \sigma_i[e].color$  then  
14           $\sigma_i[e] \leftarrow adjacent\_tile$ ;  
15        end if  
16      end for  
17    end for  
18  end if  
19 end foreach  
20 return  $\sigma_i$ 
```

---

**Scheduling loops** Using  $\Phi$ , we compute  $\sigma_i$  as described in Algorithm 3. The algorithm is similar to the projection of a tiled loop, with the main difference being that now we use  $\Phi$  to schedule iterations correctly. Finally,  $\sigma_i$  is inverted and the iterations added to the corresponding iteration lists  $T_j^{L_i}$  of all  $T_j \in \mathbb{T}$ .

**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.

### 3.6.3 The Generalized Sparse Tiling Executor

The sparse tiling executor is illustrated in Algorithm 4. It consists of four main phases: triggering of MPI communications; execution of core tiles (in overlap with communication); waiting for the termination of all MPI

---

**ALGORITHM 4:** The executor algorithm

---

**Input:** A set of tiles  $\mathbb{T}$

**Result:** Execute the loop chain, thus modifying the data sets written or incremented within  $\mathbb{L}$

```
1  $\mathbb{T}^c, \mathbb{T}^b \leftarrow \text{group\_tiles.byregion}(\mathbb{T});$ 
2  $\text{start\_MPI.communications}();$ 
3 foreach available color  $c$  do
4   | foreach  $T \in \mathbb{T}^c$  s.t.  $T.\text{color} == c$  do
5   |   |  $\text{execute\_tile}(T);$ 
6   | end foreach
7 end foreach
8  $\text{end\_MPI.communications}();$ 
9 foreach available color  $c$  do
10  | foreach  $T \in \mathbb{T}^b$  s.t.  $T.\text{color} == c$  do
11  |   |  $\text{execute\_tile}(T);$ 
12  | end foreach
13 end foreach
```

---

communications; execution of boundary tiles.

As seen in the example in Section 3.5 and based on the explanation in Section 3.6.2, we know that the core tiles do not require any off-process information to execute as long as the boundary regions are (i) up-to-date and (ii) “sufficiently deep”. If both conditions hold, the execution is semantically correct and it is safe to overlap computation of core tiles with the communication of boundary data.

**The depth of the boundary region** In  $\mathbb{L}$  we have  $n$  loops. A tile “crosses” all of these loops and is executed atomically; that is, once it starts executing its iterations, it reaches the end without the need for synchronization with other processes. With this execution model, the boundary region must include a sufficient amount of off-process iterations for a correct computation of the tiles along the border with the core region.

In the extreme case  $n = 1$ , a single “strip” of iterations belonging to adjacent processes need be redundantly executed. As  $n$  increases, more off-process data is required. If  $n = 3$ , as in the example in Figure ??,

Phase	Cost shared memory	Cost distributed memory
Partitioning	$N$	$N$
Coloring	$RK$	$N$
Projection	$R(nMNK^2)$	$nMNK^2$
Tiling	$R(nMNK)$	$nMNK$
Local maps	$nM$	$nM$

Table 3.3: Worst-case costs of inspection.

we need three “strips” of off-process iterations to be computed in order for the datasets over  $X$  to be correctly updated when executing iterations belonging to  $L_3$ .

The *depth* is a constant provided through the loop chain abstraction that informs the inspector about the extent of the boundary region. For correctness, we cannot tile more than *depth* loops, so if  $n > \text{depth}$  the loop chain must first be split into smaller sequences of loops, to be individually inspected and executed.

### 3.6.4 Computational Complexity of Inspection

Let  $N$  be the maximum size of a set in  $\mathbb{L} = [L_0, L_1, \dots, L_{n-1}]$  and let  $M$  be the maximum number of sets accessed in a loop. If  $a$  is the maximum arity of a map, then  $K = aN$  is the maximum cost for iterating over a map.  $K$  is also the worst-case cost for inverting a map. Let  $p < 1$  be the probability that a conflict arises during inspection in the case of shared memory parallelism; thus, the expected number of inspection rounds is  $R = \frac{1}{1-p}$ . Hence, the worst-case computational costs of the main inspection phases are as in Table 3.3.

## 3.7 Implementation

Our implementation of generalized sparse tiling is distributed across three software modules.

**Firedrake** This is a framework for the automated solution of finite element methods, already described in Section 2.2.2.

**PyOP2** Firedrake produces numerical kernels to be applied over sets of mesh components. The parallel iteration over the mesh is handled

by PyOP2.

**SLOPE** A library for writing generalized inspector/executor schemes, with primary focus on sparse tiling. PyOP2 uses SLOPE for tiling loop chains.

The interactions between these modules is illustrated in Figure ??.

In essence, users that want to adopt sparse tiling in their code only have to decorate a sequence of loops through a minimalistic interface exposed by PyOP2, which we refer to as the `loop_chain` interface. In the case of a Firedrake program, in which loops are automatically generated from the mathematical notation (rather than written explicitly as in pure PyOP2 program), the same result can be achieved by wrapping a critical piece of code (e.g., a time-stepping loop, in which loops will be produced) again with the `loop_chain` interface. The use of the `loop_chain` interface is exemplified in Listing ?? in two of the programs that we also employ for performance evaluation.

To overcome most of the limitations described in Section 3.2 and to achieve full automation – a feature of fundamental importance for the optimization to be actually exploited – the following strategic choices were made:

**Domain Specific Languages** Our implementation is fully integrated in a system that relies on a stack of DSLs (e.g., Firedrake and PyOP2). The abstractions provided by the DSLs extremely simplifies the analysis phase. For example, from the parallel loop construct in PyOP2 we can derive sufficient information to build a loop chain as described in Section 3.4.3.

**Frameworks with user base** We wish generalized sparse tiling to be used in real scientific software, so we had to target a system with a significant, in steady expansion, user base. This obviously increases the implementation complexity, but also maximizes the chances of seeing sparse tiling at work in multiple applications.

**Multi-layer structuring** We offer a trade-off between flexibility and implementation complexity. Adopting sparse tiling in Firedrake is the simplest option, because both the generation of inspector/executor

schemes and MPI support, as explained in Section 3.7.3, are automated. If we have a pure PyOP2 program, or we are using a DSL other than Firedrake that exploits PyOP2, the generation of inspector/executor schemes for shared memory parallelism is automated, but MPI support must be coded explicitly (in particular, the separation of a set into the core, boundary and non-exec regions). Users could also apply sparse tiling to their own legacy code adding direct calls to the SLOPE library and providing explicit MPI support, but clearly this is the most complex option.

In the next sections, we describe the main mechanisms available in these three frameworks that make automation possible.

### 3.7.1 SLOPE: a Library for Tiling Irregular Computations

SLOPE is an open source software available at <https://github.com/coneoproject/SLOPE>. It provides a set of functions to build a loop chain and to write an inspector/executor scheme for applying sparse tiling.

The loop chain abstraction supported by SLOPE was already described in Section 3.4.3. In essence, users provide the sets (including the separation into core, boundary, and non-exec regions), the relationship between them (maps), and the loops composing the loop chain. Each loop has a set of descriptors describing how the kernel accesses the values associated with some sets. The inspection is executed providing, in addition to the loop chain description, the tile size for the seed partitioning. The example in Listing 7, despite using a slightly different syntax than the original one, accurately illustrates the SLOPE's interface for inspection.

For inspection, SLOPE implements all of the algorithms in Section 3.6.2. It also provides features to verify the effectiveness and the correctness of sparse tiling, namely:

**VTK file generator** For each tiled loop, a file suitable for visualization in Paraview ? can be produced, showing the mesh and the repartition into colored tiles.

**The summary routine** Once a loop chain has been inspected, useful information are generated: the number and the average tile size, the total



number of colors used (which can give an indication of how effective a shared memory parallelization will be), times for each of the inspection phases.

In the case of shared memory parallelism, the following sections of code are parallelized through OpenMP:

- The projection and tiling algorithms; in particular, the loop at line 11 of Algorithm 2 and the loop at line 8 of Algorithm 3).
- The execution of tiles having the same color; that is, the loops at lines 4 and 10 of Algorithm 4.

### 3.7.2 PyOP2: a Runtime Library for Mesh Iteration based on Lazy Evaluation

PyOP2 has already been detailed in Section 2.2.3, so in this section we rather focus on two complementary aspects: the lazy evaluation mechanism and the interface with SLOPE.

PyOP2 can lazily evaluate parallel loops. Parallel loops encountered in the code or generated through Firedrake are pushed into a queue rather than promptly executed. The sequence of parallel loops in the queue is called *trace*. If a field  $f$  needs be read, for example because a user wants to inspect its values or a global linear algebra operation is about to be performed, the trace is traversed – from the most recently pushed parallel loop to the oldest one – and a new sub-trace produced. The sub-trace includes all parallel loops that need be executed for correctly updating  $f$ .

The `loop_chain` interface is used to identify the sub-sequence of loops in the trace that will be sparse tiled. In particular, to create an inspector/executor scheme, SLOPE acts as follows:

1. As seen in the examples in Listing ??, the `loop_chain` defines a new scope. As this scope is entered, a stamp  $s_1$  of the loop chain is generated. This happens “behind the scenes”, because the `loop_chain` is a Python context manager, which invokes pre-specified routines prior and after the execution of the body. As the `loop_chain` is exited, a new stamp  $s_2$  of the trace is computed, and all parallel loops generated between  $s_1$  and  $s_2$  are placed into a list for pre-processing.

2. The pre-processing of the extracted list of loops consists of two main phases: “simple” fusion of all consecutive parallel loops iterating over the same iteration space that do not present read-after-write or write-after-read dependencies through indirections/maps; generation of a loop chain suitable for SLOPE.
3. SLOPE provides a thin Python interface to ease integration with other frameworks. PyOP2 inspects the sequence of loops and translates data structures (sets, maps, loops) into a format suitable for the SLOPE’s Python interface, which finally returns a string of C code. This code implements an inspector for the provided loop chain. PyOP2 just-in-time compile the code and runs it, obtaining the *inspection* data structure.
4. A “software cache” mapping loop\_chains to *inspection* data structures is used. This process is therefore executed only once, because each time (e.g., in a time-stepping loop) the same loop\_chain is encountered, the corresponding *inspection* is retrieved.
5. The executor is built analogously, and invoked each time loop\_chain is encountered.

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

Firedrake uses DMPlex ? to handle meshes. In particular, DMPlex is responsible for partitioning, distributing over multiple processes, and locally reordering a mesh. The MPI parallelization, therefore, must completely be handled by Firedrake.

During the start-up phase, each MPI process receives a contiguous partition of the original mesh from DMPlex. It then creates various PyOP2 sets, representing either mesh components (e.g., cells, vertices) or function spaces. As seen in Section 2.2.3, PyOP2 sets distinguish between different regions: core, owned, exec, and non-exec. Firedrake is directly responsible for setting the halo regions and creating data structures describing the communication pattern among different processes.

For supporting our loop chain abstraction, Firedrake must be able to allocate arbitrarily deep halo regions. Firedrake and DMPlex have been

made more flexible to support this feature<sup>3</sup>. An additional parameter, called *S-depth* for historical reasons <sup>?</sup>, is now accepted when initializing a mesh. A value *S-depth* = *n* indicates the presence of *n* strips of off-process data elements; the default value is *S-depth* = 1, which implements the standard execution model in Firedrake/PyOP2 (i.e., no tiling, but a small amount of redundant computation along partition boundaries to overlap communication with computation).

## 3.8 Performance Evaluation

The experimentation was divided into two phases:

1. Generalized sparse tiling was initially applied to two benchmarks: a sparse Jacobi kernel and a proxy unstructured mesh application, originally developed as a demo for the OP2 framework. This was especially useful to identify strengths and limitations of the technique.
2. Then, it was used in an application developed in Firedrake, Seigen, an Elastic Wave Equation Solver for Seismological Problems<sup>4</sup>. The aim of this phase was to evaluate the performance improvements in a real-world simulation.

### 3.8.1 Benchmarks

In this section, our objective is to explore the impact of generalized sparse tiling on performance, to characterize the circumstances where the approach is profitable, to identify the major limitations to a widespread adoption of the technique, and to justify some of the design choices made in the previous sections.

All benchmarks are written in C. The sparse tiled implementations only support shared memory parallelism via OpenMP. A previous version of the inspector algorithms presented in this chapter, described in <sup>?</sup>, was used. As detailed next, one of the major drawbacks of this older inspector was its cost, which grew very rapidly with the number of loops in the chain and the number of distinct datasets accessed.

---

<sup>3</sup>Michael Lange carried out the bulk of the implementation.

<sup>4</sup>Seigen has been developed by Christian Jacobs.

Matrix name	Execution time reduction	Speed-up
<i>ldoor</i>	40.34	12.11
<i>pwtk</i>	38.42	11.98
<i>thermal2</i>	25.78	11.08
<i>xenon2</i>	20.15	9.53
<i>audikw.1</i>	13.42	8.70
<i>nd24k</i>	-151.72	3.06

Table 3.4: Execution time reductions over the original implementation (in percentage) and speed-ups over the single-threaded tiled implementation for the sparse Jacobi solver with 15 threads.

In all the experiments presented in this section, the optimal tile size (i.e. the one leading to the best execution time) was determined empirically, for each combination of architecture and application.

### Sparse Jacobi

The first experiment was the full sparse tiling of a Jacobi sparse matrix solver<sup>5</sup>. Given a sparse matrix  $A$ , and a vector  $\vec{f}$ , related by  $A\vec{u} = \vec{f}$ , each iteration of the sparse Jacobi method produces an approximation to the unknown vector  $\vec{u}$ . In our experiments, the Jacobi convergence iteration loop is unrolled by a factor of two and the resulting two loops are chained together (1000 iterations of the loop chain was executed). Using a ping-pong strategy, each loop reads from one copy of the  $\vec{u}$  vector and writes to the other copy. This experiment was run on an Intel Westmere (dual-socket 8-core Intel Xeon E7-4830 2.13 GHz, 24MB shared L3 cache per socket). The code was compiled using gcc-4.7.0 with options -O3 -fopenmp.

The Jacobi recurrence equation includes a sparse matrix vector multiplication and is representative of a broad class of sparse linear algebra applications. It is also an effective test-bed because different data dependency patterns can be examined simply by using different input matrices. In these experiments, a set of 6 input matrices, drawn from the University of Florida Sparse Matrix Collection<sup>?</sup>, was used. The matrices were selected so that they would vary in overall data footprint, from 45 MB to

<sup>5</sup>This section is partly extracted from<sup>?</sup>, and the results were obtained from experiments conducted by Christopher D. Krieger, Catherine Olschanowsky, and Michelle Mills Strout

Implementation	Execution time	Speed-up
Westmere <i>omp</i>	36.87	6.43
Westmere <i>mpi</i>	31.0	7.66
Westmere <i>tiled</i>	26.49	8.96
Sandy Bridge <i>omp</i>	30.01	6.65
Sandy Bridge <i>mpi</i>	24.42	8.17
Sandy Bridge <i>tiled</i>	20.63	9.67

Table 3.5: Execution time (in seconds) and speed-ups over the slowest single-threaded implementation for the Airfoil benchmark. The values are obtained from simulations with fully-loaded machines (16 and 24 threads/processes on the Sandy Bridge and the Westmere architectures, respectively).

892 MB, and in percentage of non-zeros, from very sparse at 0.0006% to much more dense at 0.5539% non-zeros.

Table 3.4 compares the performance of the tiled Jacobi solver to that of a simple blocked version. Both codes use OpenMP `parallel` for directives to achieve parallelism. The execution time reduction varied from 13% to 47% with the exception of the *nd24k* matrix, which showed as much as a 1.52x slowdown when sparse tiled. This matrix is highly connected, thus limiting the number of tiles that can be scheduled in parallel. The greater parallelism available under a blocked approach provides more benefit in this case than the performance improvements due to improved locality from full sparse tiling. Overall, speed-ups of between 8 and 12 times over the single-threaded tiled implementation were observed when using 15 threads; a clear outlier is again the *nd24k* matrix that did not scale past 3.2 times the single thread performance.

The values in Table 3.4 do not include the inspection time necessary to full sparse tile the loop chain. To break even when this cost is considered, the inspector time must be amortized over between 1000 and 3000 iterations of the executor, depending on the specific matrix being solved. We will further elaborate on this aspect in Section 3.8.1.

## Airfoil

Airfoil is a representative unstructured mesh application ?. Three implementations of Airfoil, *omp*, *mpi* and *tiled*, were compared on two shared-

memory machines, an Intel Westmere (dual-socket 6-core Intel Xeon X5650 2.66 GHz, 12MB of shared L3 cache per socket) and an Intel Sandy Bridge (dual-socket 8-core Intel Xeon E5-2680 2.00Ghz, 20MB of shared L3 cache per socket). The code was compiled using the Intel `icc 2013` compiler with optimizations enabled (`-O3, -xSSE4.2/-xAVX`).

The Airfoil code consists of a main time loop with 2000 iterations. This loop contains a sequence of four parallel loops that carry out the computation. In this sequence, the first two loops, called *adt-calc* and *res-calc*, constitute the bulk of the computation. *Adt-calc* iterates over cells, reads from adjacent vertices and write to a local dataset, whereas *res-calc* iterates over edges and exploits indirect mappings to vertices and cells for incrementing indirect datasets associated to cells. These loops share datasets associated with cells and vertices. Datasets are composed of doubles.

In the *omp* and *mpi* implementations of Airfoil, the OpenMP and the MPI back-ends of OP2, were used. The effectiveness of these parallelization schemes has been demonstrated in ?. The *tilled* implementation uses an early version of the SLOPE library (the differences with the inspector algorithms shown in Section 3.6.2 are discussed later) for tiling a loop chain. We manually unrolled the time loop by a factor of two to be able to tile over 6 loops in total.

Table 3.5 shows the scalability and runtime reduction achieved by sparse tiling the loop chain on the Westmere and Sandy Bridge architectures. The input unstructured mesh was composed of 1.5 million edges. It is worth noticing that both the *omp* and *tilled* versions suffer from the well-known NUMA effect as threads are always equally spread across the two sockets. Nevertheless, compared to *mpi*, the *tilled* version exhibits a peak runtime reduction of 15% on the Westmere and of 16% on the Sandy Bridge.

Results shown for *tilled* do not include, however, the inspector overhead. By also including it, the aforementioned improvements over *mpi* reduce to roughly 10% on both platforms. Similarly to the sparse Jacobi solver, the slow-downs when including the inspection overhead are significant.

## Observations

One of the most important outcomes of this first set of experiments concerns the inspection cost, which we found to significantly impact the over-

all execution time. To effectively support real applications like Seigen, typically characterized by a number of parallel loops larger than that of these benchmarks, we had to rework the original inspector algorithms into the versions described in Section 3.6.2. The most notable differences are: (i) data dependency analysis abstracted to the level of sets, rather than datasets; (ii) optimistic coloring with backtracking in case of conflicts; (iii) fully parallel projection and tiling phases, through the use of inverse maps.

A second observation is about the importance of automation. Writing the inspector as a sequence of calls to SLOPE is relatively simple, although tedious and error-prone. Much more complicated is integrating the executor, because this essentially means rewriting entire sequences of loops – a severe limitation when using SLOPE in plain C (perhaps legacy) code. These considerations on the implementation complexity led to the multi-layer framework detailed in Section 3.7 for automating the generation of inspector/executor schemes.

The Airfoil experiments highlighted that shared memory parallelism over multi-socket architectures is considerably affected by the NUMA issue. The difference in execution time between the OP2 OpenMP and MPI versions is indeed remarkable. As discussed in ?, the irregular nature of the computation makes it hard to find systematic solutions to the NUMA problem when relying on OpenMP. The MPI and the hybrid OpenMP/MPI execution models are simply better candidates for unstructured mesh computations. The sparse tiled implementation, which in these experiments is also purely based on OpenMP, suffers from the NUMA issue as well, although it manages to outperform the OP2 MPI version thanks to the relieved memory pressure. We wondered, however, what the gain would be if we managed to integrate MPI with sparse tiling. This led to the definition of more general loop chain abstraction (Section 3.4.3) and inspector algorithms (Section 3.6.2), for supporting MPI-based executors.

### **3.8.2 Seigen: an Elastic Wave Equation Solver for Seismological Problems**

...

### **3.9 Conclusions and Future Work**



## **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 prob-

lems 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 unit-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  $3 \times 3$  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  $3 \times 4$ . 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 1 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 1:** 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.

---

```

1 void weighted_laplace(double A[3][3], double **coords, double w[3]) {
2     #define ALIGN __attribute__((aligned(32)))
3     // K, det = Compute Jacobian (coords)
4
5     // Quadrature weights
6     static const double W[6] ALIGN = 0.5;
7
8     // Basis functions
9     static const double B[6][4] ALIGN = {{...}} ;
10    static const double C[6][3] ALIGN = {{...}} ;
11    static const double D[6][4] ALIGN = {{...}} ;
12
13    // Padded buffer
14    double _A[3][4] ALIGN = {{0.0}};
15
16    for (int i = 0; i<6; i++) {
17        double f0 = 0.0;
18        for (int r = 0; r < 3; ++r) {
19            f0 += (w[r] * C[i][r]);
20        }
21        double T_0[4] ALIGN;
22        double T_1[4] ALIGN;
23        #pragma vector aligned
24        for (int k = 0; k<4; r++) {
25            T_0[r] = ((K[1]*B[i][k])+(K[3]*D[i][k]));
26            T_1[r] = ((K[0]*B[i][k])+(K[2]*D[i][k]));
27        }
28        for (int j = 0; j<3; j++) {
29            #pragma vector aligned
30            for (int k = 0; k<4; k++) {
31                _A[j][k] += (T_0[k]*T_0[j] + T_1[k]*T_1[j])*det*W[i]*f0);
32            }
33        }
34    }
35 }
36 for (int j = 0; j<3; j++) {
37     for (int k = 0; k<3; k++) {
38         A[j][k] = _A[j][k];
39     }
40 }

```

---

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

---

**LISTING 2:** 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.

---

```

1 void weighted_laplace(double A[3][3], double **coords, double w[3]) {
2   // Omitting redundant code
3   ...
4   for (int j = 0; j<3; j++) {
5     for (int k = 0; k<3; k++) {
6       A[j][k] += (T_0[k]*T_0[j])*det*W[i]*f0;
7     }
8   }
9   for (int j = 0; j<3; j++) {
10    for (int k = 0; k<3; k++) {
11      A[j][k] += (T_1[k]*T_1[j])*det*W[i]*f0;
12    }
13  }
14 }
15 ...

```

---

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  $\text{det} * W3[i]$  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 3:** 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.

---

```

1 void weighted_laplace(double A[3][3], double **coords, double w[3]) {
2     // Omitting redundant code
3     ...
4     // Padded buffer (note: both rows and columns)
5     double _A[4][4] ALIGN = {{0.0}};
6
7     for (int i = 0; i<3; i++) {
8         // Omitting redundant code
9         // ...
10        for (int j = 0; j<4; j += 4)
11            for (int k = 0; k<4; k += 4) {
12                // Sequence of LOAD and SET intrinsics
13                // Compute _A[0][0], _A[1][1], _A[2][2], _A[3][3]
14                // One _mm256_permute_pd per k-loop LOAD
15                // Compute _A[0][1], _A[1][0], _A[2][3], _A[3][2]
16                // One _mm256_permute2f128_pd per k-loop LOAD
17                // ...
18            }
19        // Scalar remainder loop (not necessary in this example)
20    }
21    // Restore the storage layout
22    for (int j = 0; j<4; j += 4) {
23        for (int k = 0; k<4; k += 4) {
24            _mm256d r0 = _mm256_load_pd (&_A[j+0][k]);
25            // LOAD _A[j+1][k], _A[j+2][k], _A[j+3][k]
26            r4 = _mm256_unpackhi_pd (r1, r0);
27            r5 = _mm256_unpacklo_pd (r0, r1);
28            r6 = _mm256_unpackhi_pd (r2, r3);
29            r7 = _mm256_unpacklo_pd (r3, r2);
30            r0 = _mm256_permute2f128_pd (r5, r7, 32);
31            r1 = _mm256_permute2f128_pd (r4, r6, 32);
32            r2 = _mm256_permute2f128_pd (r7, r5, 49);
33            r3 = _mm256_permute2f128_pd (r6, r4, 49);
34            _mm256_store_pd (&_A[j+0][k], r0);
35            // STORE _A[j+1][k], _A[j+2][k], _A[j+3][k]
36        }
37    }
38 }
39 ...

```

---

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

$$A_{jk} = \sum_{\substack{x \in B' \subseteq B \\ y \in B'' \subseteq B}} x_j \cdot y_k \quad j, k = 0, \dots, 2 \quad (5.1)$$

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 ( $j, k = 0, \dots, 3$ ) of the element matrix using just 2 vector registers, which represent a  $4 \times 4$  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 3.



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

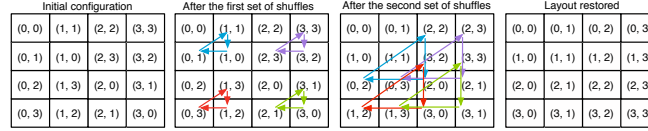


Figure 5.2: Restoring the storage layout after *op-vect*. The figure shows how  $4 \times 4$  elements in the top-left block of the element matrix  $A$  can be moved to their correct positions. Each rotation, represented by a group of three same-colored arrows, is implemented by a single shuffle intrinsic.

## 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 prob-

lems, 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 ( $X1, X2, \dots$ ) and constants ( $F0, F1, \dots, K0, K1, \dots$ ) 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  $30 \times 30$ , with a peak of  $105 \times 105$  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  $1.4\times$  (Diffusion, prismatic mesh,  $q = 4$  - increasing the overall speed up from  $2.69\times$  to  $3.87\times$ ).

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  $1.4\times$  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 performance. The transformation negligibly affected register locality, so speed ups up to  $1.5\times$  were observed. For instance, when  $q = 4$  and a prismatic mesh is employed, the overall performance improvement increases from  $1.44\times$  to  $2.11\times$ .

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  $1.22\times$ , *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 heuristic is 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  $10 \times 10 \times 10$ , for which extensive unrolling would already be harmful, are in practice very common. Secondly, manual unrolling is dangerous because it may compromise compiler auto-vectorization by either removing loops (most compilers search for vectorizable loops) or losing spatial locality within a vector register.

By comparison 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 4:** The assembly kernel for the weighted Laplace operator in Listing 1 after application of vector promotion (on top of generalized code motion).

---

```

1 void weighted_laplace(double A[3][3], double **coords, double w[3]) {
2   // Omitting redundant code
3   ...
4   double f0[3] = {0.0};
5   for (int i = 0; i < 6; i++) {
6     for (int r = 0; r < 3; ++r) {
7       f0[i] += (w[r] * C[i][r]);
8     }
9   }
10  for (int i = 0; i < 6; i++) {
11    double T_0[3] ALIGN;
12    double T_1[3] ALIGN;
13    for (int k = 0; k < 3; ++k) {
14      T_0[r] = ((K[1]*B[i][k])+(K[3]*D[i][k]));
15      T_1[r] = ((K[0]*B[i][k])+(K[2]*D[i][k]));
16    }
17    for (int j = 0; j < 3; ++j) {
18      for (int k = 0; k < 3; ++k) {
19        A[j][k] += (T_0[k]*T_0[j] + T_1[k]*T_1[j])*det*w[i]*f0[i]);
20      }
21    }
22  }
23 }
```

---

Consider Listing 4. The evaluation of the coefficient  $w$  at each quadra-

ture 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 in another computational domain, 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

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

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  $\alpha$ ,  $\beta$  and  $\gamma$  are problem-specific and can be intricately complex, involving for example the evaluation of derivatives. We can however abstract from the inherent structure of  $\alpha$ ,  $\beta$  and  $\gamma$  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  $\beta$  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  $\alpha$ ,  $\beta$  and  $\gamma$  be split (assuming they include associative operators), but  $\alpha$  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<sup>1</sup>, a mature, platform-agnostic compiler. COFFEE has fully been integrated with Firedrake, the framework based on the finite element method introduced in Section 2.2.2. 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) <sup>2</sup>, 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.

---

<sup>1</sup>COFFEE is the acronym for COmpiler For Fast Expression Evaluation.

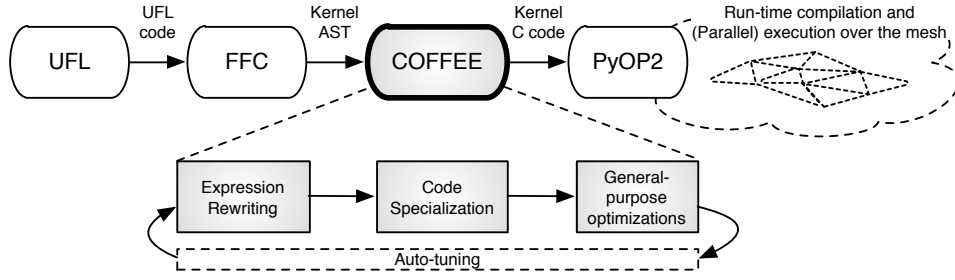


Figure 6.1: High-level view of Firedrake. COFFEE is at the core, receiving ASTs from a modified version of the FEniCS Form Compiler and producing optimized C code kernels.

## 6.2 The Compilation Pipeline

Similarly to general-purpose compilers, COFFEE provides different optimization levels, namely 00, 01, 02 and 03. Apart from 00, 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 01, 02 and 03, we will use the generic notation  $0x$  ( $x \in \{1, 2, 3\}$ ).

The optimization level  $0x$  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 `0x`. 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 `0x`, 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:

- 01** 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<sup>2</sup>. Another special type of node is `ArrayInit`, used for static initialization of arrays. An `ArrayInit` wraps an `N`-dimensional Numpy array<sup>?</sup> 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

---

<sup>2</sup>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.

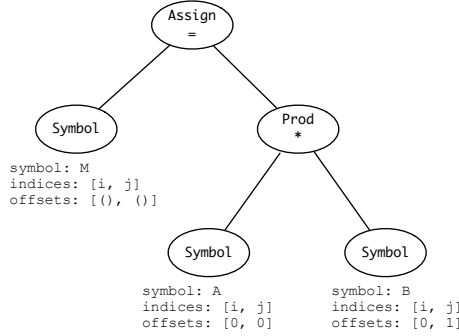
$$M[i][j] = A[i][j] * B[i][k+1]$$


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

entry  $r$  in the rank, there is a corresponding entry  $\langle scale, stride \rangle$  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)<sup>3</sup>. These

<sup>3</sup>The generation of ASTs in TSFC has been written by Myklos Homolya.

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.

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<sup>4</sup>, 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

---

<sup>4</sup>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 01); 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. Lorient, 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. Lorient, 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>.