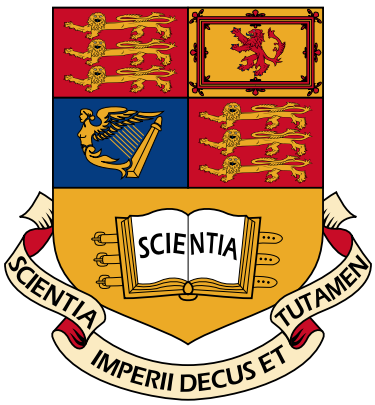
Imperial College London
Department of Computing

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

Chapter 1

Introduction

1.1 Thesis Statement

1.2 Overview

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

The time required to execute these unstructured-mesh-based applications is a fundamental issue. An equation domain needs to be discretized into an extremely large number of cells to obtain a satisfactory approximation of the solution, possibly of the order of trillions (e.g. Rossinelli et al. [2013]), so applying numerical kernels all over the mesh is expensive. For

example, it is well-established that mesh resolution is crucial in the accuracy of numerical weather forecasts; however, operational centers have a strict time limit in which to produce a forecast - 60 minutes in the case of the UK Met Office - so, executing computation- and memory-efficient kernels has a direct scientific payoff in higher resolution, and therefore more accurate predictions. Motivated by this and analogous scenarios, this thesis studies, formalizes, and implements a number of code transformations to improve the performance of real-world scientific applications using numerical methods over unstructured meshes.

1.3 Contributions

1.4 Dissemination

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

- **Papers.** The following is the list of publications derived from the research activity (chronological order):

  1. Strout, M.M.; Luporini, F.; Krieger, C.D.; Bertolli, C.; Bercea, G.-T.; Olschanowsky, C.; Ramanujam, J.; Kelly, P.H.J., "Generalizing Run-Time Tiling with the Loop Chain Abstraction," Parallel and Distributed Processing Symposium, 2014 IEEE 28th International , vol., no., pp.1136,1145, 19-23 May 2014

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

  3. Fabio Luporini, David A. Ham, Paul H. J. Kelly. "Optimizing Automated Finite Element Integration through Expression Rewriting and Code Specialization". 2014. To be written.

- **Talks.** Talks have been delivered at the following conferences/workshops:

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

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

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

1.5 Thesis Outline

Chapter 2

Background

2.1 The Finite Element Method

Computational methods based upon finite elements are used to approximate the solution of partial differential equations (henceforth, PDEs) in a wide variety of domains. The mathematical abstraction used in finite element methods (or FEMs) is extremely powerful: not only does it help reasoning about the problem, but also provides systematic ways of deriving effective computer implementations. In **?**, it is usefully suggested to consider an FEM as a black box that, given a differential equation, returns a discrete algorithm capable of approximating the equation solution. Unveiling the whole magic of such a black box is clearly out of the scope of this chapter. We rather limit ourselves to review the mathematical properties and the computational aspects that are essential for understanding the contributions in Chapters 4 and 5. The content and the notation used in this section are inspired from **???**. For a complete treatment of the subject, the reader is invited to refer to **?**.

2.1.1 Variational Formulation

We consider the weak formulation of a *linear variational problem*

$$\begin{gathered} \text{Find } u \in U \text{ such that} \\ a(u,v) = L(v), \forall v \in V \end{gathered} \tag{2.1}$$

where a and L are, respectively, a bilinear form and a linear form. The term "variational" stems from the fact that the function v can vary arbitrarily. The unfamiliar reader may find this expression unusual. Understanding the actual meaning of this formulation is far beyond the goals of this review, since an entire course in functional analysis would be needed. Informally, we can think of V as a "very nice" space, in which functions have desirable properties. The underlying idea of the variational formulation is to "transfer" certain requirements (e.g., differentiability) from the unknown $u \in U$ to $v \in V$.

The sets U and V and called, respectively, trial and test functions. The variational problem is discretized by using discrete test and trial spaces

$$\begin{aligned} &\text{Find } u_h \in U_h \subset U \text{ such that} \\ &a(u_h, v_h) = L(v_h), \forall v_h \in V_h \subset V \end{aligned} \tag{2.2}$$

Let $\{\psi_i\}_{i=1}^N$ be the set of basis functions spanning U_h and let $\{\phi_j\}_{j=1}^N$ be the set of basis functions spanning V_h. Then the unknown solution u can be approximated as a linear combination of the basis functions $\{\psi_i\}_{i=1}^N$,

$$u_h = \sum_{j=1}^{N} U_j \psi_j. \tag{2.3}$$

This allows us to rewrite 2.2 as:

$$\sum_{j=1}^{N} U_j a(\psi_j, \phi_i) = L(\phi_i),\ i = 1, 2, ..., N \tag{2.4}$$

From the solution of the following linear system we determine the set of *degrees of freedom* U to express u_h:

$$Au = b \tag{2.5}$$

where clearly

$$\begin{aligned} A_{ij} &= a(\phi_i(x), \phi_j(x)) \\ b_i &= L(\phi_i(x)) \end{aligned} \tag{2.6}$$

The matrix A and the vector b can be seen as the discrete operators arising from the bilinear form a and from the linear form L for the given choice of basis functions.

The variational formulation of a *non-linear variational problem* requires refinements that are out of the scope of this review. The interested reader should refer to **?**.

2.1.2 Finite Elements

In an FEM the domain Ω of the PDE is partitioned into a finite set of disjoint cells $\{K\}$; that is, $\bigcup K = \Omega$ and $\bigcap K = \emptyset$. This forms a *mesh*. A finite element is usually defined (see for example **?**) as a triple $<K, \mathcal{P}_K, \mathcal{L}_K>$, where:

- K is a cell in the mesh with non-empty interior and piecewise smooth boundary;
- $\mathcal{P}_K$ is a finite dimensional "local" function space of dimension n_K;
- the set of degrees of freedom $\mathcal{L}_K$ is a basis $\{l_1^K, l_2^K, ..., l_{n_K}^K\}$ for $\mathcal{P}_K'$, the dual space of $\mathcal{P}_K$.

This definition allows to impose constraints on the set of basis functions $\{\phi_1^K, \phi_2^K, ..., \phi_{n_K}^K\}$ spanning $\mathcal{P}_K$. For instance, to enforce a *nodal basis* for $\mathcal{P}_K$ – a particularly useful property for expressing solutions in U_h – we can impose that the relationship

$$l_i^K(\phi_j^K) = \delta_{ij},\ i, j = 1, 2, ..., n_K \tag{2.7}$$

must be satisfied (δ_{ij} is the Kronecker delta). This allows to express any $v \in \mathcal{P}_K$ as

$$v = \sum_{i=1}^{n_K} l_i^K(v)\phi_i^K. \tag{2.8}$$

Each linear functional in $\mathcal{L}_K$ is used to evaluate one degree of freedom of v in terms of the chosen nodal basis. In other words, we can refer to both the coefficients U introduced in the previous section and $\mathcal{L}_K$ as the degrees of freedom.

Example: the triangular Lagrange element As an example[1], consider a triangular cell K and let $\mathcal{P}_K$ be the space of polynomials of order 1 on K. $\mathcal{L}_K$ may be the set of bounded linear functionals representing point evaluation at the vertices $\boldsymbol{x}^i$ $(i = 1, 2, 3)$ of K such that

$$\begin{aligned} l_i^K &: \mathcal{P}^{\mathcal{K}} \to \mathbb{R} \\ l_i^K(v) &= v(\boldsymbol{x}^i) \end{aligned} \tag{2.9}$$

Since if v is zero at each vertex then v must be zero everywhere, $\mathcal{L}_K$ really is a basis for $\mathcal{P}_K'$, so what we have defined is indeed a finite element. In particular, if we take $\boldsymbol{x}^1 = (0,0)$, $\boldsymbol{x}^2 = (1,0)$, $\boldsymbol{x}^3 = (0,1)$, we have that the nodal basis is given by:

$$\phi_1(\boldsymbol{x}) = 1 - x_1 - x_2, \quad \phi_2(\boldsymbol{x}) = x_1, \quad \phi_3(\boldsymbol{x}) = x_2. \tag{2.10}$$

2.1.3 Global Discretization

A *local-to-global mapping* allows to patch together the finite elements to form a global function space, for instance the set of trial functions $U_h = \text{span}\{\psi_i\}_{i=1}^N$ introduced in Section 2.1.1. A local-to-global mapping is a function

$$\iota_K : [1, n_K] \to [1, N] \tag{2.11}$$

that maps the local degrees of freedom $\mathcal{L}_K$ to global degrees of freedom $\mathcal{L}$. The mappings ι_K, together with the choice of $\mathcal{L}_K$, determine the continuity of a function space or, in simpler words, the continuity of a function throughout the domain Ω. The reader is invited to refer to **?** for a comprehensive yet simple description of this step.

One of the crucial aspects of an FEM is that global function spaces are often defined in terms of a *reference finite element* $<\hat{K}, \hat{\mathcal{P}}, \hat{\mathcal{L}}>$ and a set of invertible mappings $\{\mathcal{G}_K\}_K$ from $\hat{K}$ to each cell in the mesh such that $K = \mathcal{G}_K(\hat{K})$. This situation is illustrated in Figure **??**.

For each K, $\mathcal{G}_K$ also allows to generate $\mathcal{P}_K$ and $\mathcal{L}_K$. The complexity of this process depends on the mapping itself. In the simplest case, the

[1] The example is extracted from **?**

mapping is affine; that is, expressible as $\mathcal{G}_K(\hat{\boldsymbol{x}}) = A_K\hat{\boldsymbol{x}} + b_K$, where A_K and b_K are, respectively, some matrix and vector.

2.1.4 Assembly

The *assembly* of an FEM is the phase in which the matrix A and the vector b from 2.6 are constructed. This is accomplished by adding the contributions from each K to A and b. Let us consider the bilinear form a, and assume this is an integral over Ω. Thanks to the linearity of the operator, we can express a as

$$a = \sum_K a_K \tag{2.12}$$

where a_K is an element bilinear form. We can then define the local element matrix

$$A_i^K = a_K(\psi_{i_1}^K, \phi_{i_2}^K) \tag{2.13}$$

where $i \in \mathcal{I}_K$ is the index set on A_i^K. That is, $\mathcal{I}_K = \{(1,1),(1,2),...,(n_U,n_V)\}$, with n_U and n_V representing the number of degrees of freedom for the local trial functions $\psi^K \in U_h^K$ and the local test functions $\phi^K \in V_h^K$. The element matrix A^K is therefore a (typically dense) matrix of dimension $n_U \times n_V$.

Now let ι_K^U and ι_K^V be the local-to-global mappings for the local discrete function spaces U_h^K and V_h^K. We can define, for each K, the collective local-to-global mapping $\iota_K : \mathcal{I}_K \to \mathcal{I}$ such that

$$\iota_K(i) = (\iota_K^U(i_1), \iota_K^V(i_2)) \ \forall i \in \mathcal{I}_K. \tag{2.14}$$

This simply maps a pair of local degrees of freedom to a pair of global degrees of freedom. Let also $\mathcal{T}$ be the subset of the cells in the mesh in which ψ_{i_1} and ϕ_{i_2} are both non-zero. Note that here we are talking about the global functions whose restrictions to K gives $\psi_{i_1}^K$ and $\phi_{i_2}^K$. By construction, ι_K is invertible if $K \in \mathcal{T}$. At this point, we have all the ingredients to compute A as the sum of local contributions from the cells

in the mesh:

$$\begin{aligned} A_i &= \sum_{K \in \mathcal{T}} a_K(\psi_{i_1}, \phi_{i_2}) \\ &= \sum_{K \in \mathcal{T}} a_K(\psi^K_{(\iota_K^U)^{-1}(i_1)}, \phi^K_{(\iota_K^V)^{-1}(i_2)}) = \sum_{K \in \mathcal{T} A^K_{\iota_K^{-1}(i)}} \end{aligned} \tag{2.15}$$

Similar conclusions may be drawn for the linear form L. We observe that this computation can be implemented as a single iteration over all cells in the mesh. On each cell, A^K is computed and added to A using the inverse mappings. This approach is particularly efficient because it only evaluates the non-zero entries in the sparse matrix A. Other more trivial implementations of the assembly phase are possible, although rarely used in practice because ineffective.

We conclude with a clarification concerning the terminology. The assembly process is often naturally described as a two-step procedure: *local assembly* and *global assembly*. The former consists of computing the contributions of each single element (i.e., the element matrices A^K); the latter represents the "coupling" of all A^K into A. As we shall see, one of the main subjects of this thesis is the computational optimization of the local assembly phase.

2.1.5 Local Assembly Example: from Math to Code

Because of its relevance in this thesis, we illustrate local assembly in a concrete example, the evaluation of the element matrix for a Laplacian operator.

Specification of the Laplacian operator

Consider the weighted Laplace equation

$$-\nabla \cdot (w \nabla u) = 0 \tag{2.16}$$

in which u is unknown, while w is prescribed. The bilinear form asso-

ciated with the weak variational form of the equation is:

$$a(v,u) = \int_\Omega w \nabla v \cdot \nabla u \, \mathrm{d}x \tag{2.17}$$

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

$$A_{ij}^K = \int_K w \nabla \phi_i^K \cdot \nabla \phi_j^K \, \mathrm{d}x \tag{2.18}$$

The element vector L can be determined in an analogous way.

In this example, the element tensors are expressed as a single integral over the cell domain. In general, they are expressed as a sum of integrals over K, each integral being the product of derivatives of functions from sets of discrete spaces and, possibly, functions of some spatially varying coefficients. Such an integral is often called *monomial*.

Quadrature Mode

A quadrature scheme is typically used to numerically evaluate A_{ij}^K. It consists of evaluating an integral at a set of given *quadrature points*, each point multiplied with some suitable *quadrature weight*. By mapping the computation to a reference element as explained in Section 2.1.3 and using the same approach as in Section 2.1.4, a quadrature scheme for the Laplacian operator on K is as follows

$$\begin{aligned} A^K &= \int_K w \nabla \phi_i^K \cdot \nabla \phi_j^K \\ &\approx w \sum_{k=1}^{N_q} W_k \nabla \phi_i^K(\boldsymbol{x}^k) \cdot \nabla \phi_i^K(\boldsymbol{x}^k) |\det \mathcal{G}_K'(\boldsymbol{x}^k)|, \end{aligned} \tag{2.19}$$

where $\{\boldsymbol{x}^1, \boldsymbol{x}^2, ..., \boldsymbol{x}^{N_q}$ is the set of N_q quadrature points, and $\{W_1, W_2, ..., W_{N_q}\}$ the corresponding set of quadrature weights scaled such that $\sum_{k=1}^{N_q} W_k = |\hat{K}|$.

To compute a local basis function ϕ_i^K from a reference element basis

function Φ_i we exploit the invertible map $\mathcal{G}_K^{-1}$, which allows us to write $\phi_i^K = \Phi_i \circ \mathcal{G}_K^{-1}$. To evaluate the gradient of a basis function ϕ_i^K at a quadrature point $\boldsymbol{x}^k$, with $\boldsymbol{x}^k = \mathcal{G}_K(\boldsymbol{X}^k)$ and $\boldsymbol{X}^k \in \hat{K}$, we therefore have to compute a matrix-vector product

$$\nabla_x \phi_i^K(\boldsymbol{x}^k) = ((\mathcal{G}_K')^{-1})^T(\boldsymbol{x}^k) \nabla_X \Phi_i(\boldsymbol{X}^k). \tag{2.20}$$

The term $\mathcal{G}_K')^{-1}$ represents the inverse of the Jacobian matrix originating from the change of coordinates. The resulting scalar-valued expression for each entry A_{ij}^K, assuming Ω to be a domain of dimension d, reads as:

$$A_{ij}^K = \sum_{k=1}^{N_q} \sum_{\alpha_3=1}^{n} \phi_{\alpha_3}(\boldsymbol{X}^k) w_{\alpha_3} \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial \phi_i^K(\boldsymbol{X}^k)}{\partial X_{\alpha_1}} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \frac{\partial \phi_j^K(\boldsymbol{X}^k)}{\partial X_{\alpha_2}} \det \mathcal{G}_K' W^k. \tag{2.21}$$

Tensor Contraction Mode

Consider the case in which $\mathcal{G}_K : \hat{K} \to K$ is an affine mapping. Starting from Equation 2.21 and exploiting linearity, associativity and distributivity of the involved mathematical operators, we can rewrite 2.21 as

$$A_{ij}^K = \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\alpha_3=1}^{n} \det \mathcal{G}_K' w_{\alpha_3} \sum_{\beta=1}^{d} \frac{X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_{K_0} \phi_{\alpha_3} \frac{\partial \phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \phi_{i_2}}{\partial X_{\alpha_2}} dX. \tag{2.22}$$

A generalization of this transformation has been proposed in **?**. Because of only involving reference element terms, the integral in the equation can be pre-evaluated and stored in temporary variables. The evaluation of the local tensor can then be abstracted as

$$A_{ij}^K = \sum_{\alpha} A^0_{i_1 i_2 \alpha} G_K^\alpha \tag{2.23}$$

in which the pre-evaluated "reference tensor" $A_{i_1 i_2 \alpha}$ and the cell-dependent "geometry tensor" G_K^α are exposed.

Qualitative comparison

Depending on form and discretization, the relative performance of the two modes, in terms of the operation count, can vary quite dramatically. The presence of derivatives or coefficient functions in the input form increases the rank of the geometry tensor, making the traditional quadrature mode preferable for "complex" forms. On the other hand, speed-ups from adopting tensor mode can be significant in a wide class of forms in which the geometry tensor remains "sufficiently small". The discretization, particularly the relative polynomial order of trial, test, and coefficient functions, also plays a key role in the resulting operation count.

These two modes have been implemented in the FEniCS Form Compiler **?**, which we review in later sections. In this compiler, a heuristic is used to choose the most suitable mode for a given form. It consists of analysing each monomial in the form, counting the number of derivatives and coefficient functions, and checking if this number is greater than a constant found empirically (Logg et al. [2012]). In Chapter 4, we will describe a code generation system that goes beyond the dichotomy between quadrature and tensor modes, relying on cost models comparing the impact of different rewrite operators (e.g., expansion of products, factorization).

Code Examples

A possible C implementation of Equation 2.21 is illustrated in Listing 1. We assume a domain of dimension $d = 2$ and polynomial order 1 Lagrange elements. The values at the quadrature points of the derivatives of the basis functions are pre-tabulated in the `B` and `D` arrays (representing, respectively, the derivatives with respect to the coordinates x and y). The values at the quadrature points of the basis functions spanning the field w are pre-tabulated in the array `C`. Pre-tabulation, which is made possible by mapping the computation to a reference element, is of fundamental importance to speed-up the local assembly phase. The summation over the $N_q = 6$ quadrature points is implemented by the `i` loop. The `r` loop implements the summation over α_3 for discretization of the weight w. Given their small range, the summations over the spatial dimensions α_1, α_2 and β have all been expanded in the "assembly expression" that evaluates the

local element matrix A. The K array includes the four components of the inverse of the Jacobian matrix for the change of coordinates.

LISTING 1: A possible implementation of Equation 2.21 assuming a 2D triangular mesh and polynomial order 1 Lagrange basis functions.

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

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

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

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

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

The evaluation of integrals becomes more computationally expensive if the complexity of a variational form grows, in terms of number of coefficients and tensor algebra or differential operators employed. It is not pathological a scenario in which the computation of a local element tensor requires more than hundreds or even thousands of floating point operations. An excerpt from one such example is shown in Listing 2: here, in the main expression, 14 unique arrays are accessed (with the same array referenced multiple times within the expression) along with several other constants. The loop trip counts are also larger due to the different domain discretization employed. As we shall see, automated code generation for

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

```
void burgers(double A[12][12], double **coords, double **w)
{
  // Compute Jacobian
  double J[9];
  compute_jacobian_triangle_3d(J, coords);

  // Compute Jacobian inverse and determinant
  double K[9];
  double detJ;
  compute_jacobian_inverse_triangle_3d(K, detJ, J);
  const double det = fabs(detJ);

  // Quadrature weights
  static const double W[5] = {...}

  // Basis functions
  static const double B[5][12] = {{...}}
  static const double C[5][12] = {{...}}
  //11 other basis functions definitions
  ...
  for (int i = 0; i<5; i++) {
    double f0 = 0.0;
    //10 other declarations (f1, f2,...)
    ...
    for (int r = 0; r<12; r++) {
      f0 += (w[r][0]*C[i][r]);
      //10 analogous statements (f1, f2, ...)

    }
    for (int j = 0; j<12; j++) {
      for (int k = 0; k<12; k++) {
        A[j][k] += ...
   + ((K[5]*F9) + (K[8]*F10))*Y1[i][j]) + ...  +
   + (((K[0]*C[i][k]) + (K[3]*D[i][k]) + (K[6]*E[i][k]))*Y2[i][j]))*f11) +
   + (((K[2]*C[i][k]) + (K[5]*D[i][k]) + (K[8]*E[i][k]))*((K[2]*E[i][j]) + ...)))  +
   + <roughly a hundred sum/muls go here>)..)*det*W[i];
      }
    }
  }
}
```

finite element operators has had two main objectives:

- relieving the implementation burden when it comes to translate into code non-trivial operators, a tedious, error-prone task;
- facilitating the introduction of techniques to reduce the operation count and powerful low-level optimizations.

The second point, as already anticipated, is one of the topics treated in this thesis.

2.1.6 Linear Solvers

The last step of an FEM is the resolution of the linear system (2.5) arising from the variational form of the input problem. This and the assembly of A and b are the most expensive phases of an FEM. There is a whole science concerning the efficient resolution of linear systems. Among the most effective solvers are the family of *Krylov-type iteration methods*, such *conjugate gradient* for symmetric positive-definite matrices and *generalized minimal residual* (GMRES), which does not require A in explicit form. *Multi-grid* methods are also widely used, whereas *direct methods* computing an LU factorization using *Gaussian elimination* have limited applicability.

The resolution of linear systems is not one of the topics of this thesis, so the interested reader is invited to refer to **?**. It is however important to keep in mind that this phase usually has a significant impact on the execution time of an FEM: the performance optimization of the assembly phase has marginal impact if the method is solver-dominated.

2.2 Abstractions in Computational Science

This thesis centres on performance optimizations for scientific codes targeting different layers of abstraction. In this section, we dive into such abstractions and review the state-of-the-art on established software. The objective is to provide a foundation upon which motivating and explaining the contributions in Chapters 3, 4 and 5.

2.2.1 Automating the Finite Element Method

The need for rapid implementation of high performance, robust, and portable scientific simulations has led to approaches based on automated code generation. This has been proven extremely successful in the context of the FEniCS (Logg et al. [2012]) and Firedrake (**?**) projects, which target the finite element method. In these frameworks, the weak variational form of a problem is expressed at high level by means of a domain-specific language. The mathematical specification is then manipulated by a form compiler that generates a representation of the assembly operators. The representation is transformed for performance optimization and subsequently translated into C code. The generated code is compiled and executed over the mesh. This entire process occurs at run-time because both FEniCS and Firedrake are implemented in Python and rely on just-in-time code generation and compilation. When the operators are assembled, a linear system needs be solved. For this, the PETSc library **?** is employed. In the following, we describe the aspects of this tool-chain that are relevant for the following chapters. We focus on Firedrake, rather than FEniCS, because all algorithms and techniques developed in this thesis have been integrated with this framework.

Problem Specification

Firedrake uses a mathematical language called UFL, the *Unified Form Language* **?**. UFL is unrelated to meshes, function spaces, and solvers. It only concerns with the variational formulation of a problem, by providing different kinds of operators – like `grad` (gradient) and `inner` (inner product), but also algebraic operators (e.g., `transpose`, `inverse`) and elementary functions (e.g., `abs`, `sqrt`). Before emitting a representation suitable for the underlying compiler, UFL analyzes the form to collect useful information and applies some preliminary transformations, including automatic differentiation.

A UFL representation of the weighted Laplace operator shown in (2.17) is given in Listing 3. When constructing a finite element, three pieces of information are specified: *family*, *cell*, and *polynomial degree*. The family allows varying the type of the element. UFL supports many types, ranging from the most standard *Lagrange* element to *Discontinuous Lagrange*

LISTING 3: UFL specification of the weighted Laplace operator defined in (2.17). In orange the keywords of the language.

```
element = FiniteElement ('Lagrange', triangle, 1)

u = TrialFunction (element)
v = TestFunction (element)
w = Coefficient (element)

a = w*dot (grad(v), grad(u))*dx
```

and even mixed elements such as *H(div)* and *H(curl)*. This allows solving problems with the most disparate requirements on continuity of the functions. A thorough description is provided in **?**. The cell represents the shape of the reference element: possible values include *triangle*, *quadrilateral* and *tetrahedron*. The polynomial order drives the number of degrees of freedom in an element. Functions can also be vector-valued, in which case one must use the special construct *VectorElement* in place of *FiniteElement*.

We are not interested in a complete review of UFL, which would also require a knowledge of the finite element method that goes far beyond the tool-kit we need in this thesis. It is however worth appreciating the power, versatility and depth of this language, since it opened a new era in the development of computational methods.

Form Compilers

The UFL specification of a variational form is eventually passed to a form compiler, whose objective is to construct a representation of the assembly operators. The *FEniCS Form Compiler*, or FFC, was originally adapted and used in Firedrake for this task. FFC, which supports both quadrature and tensor modes (see Section 2.1.5), used to directly emit C code. It was later modified (one of the initial steps of this thesis) to rather output abstract syntax trees, a structure allowing further manipulation in the lower layers of the stack. More recently, FFC has been supplanted by a new system, the *Two-Stage Form Compiler*, or TSFC. Just like FFC, TSFC emits abstract syntax trees (ASTs). The optimizations described in Chapters 4 and 5 are implemented by manipulating the AST representation in a lower-level compiler, COFFEE (outlined in Chapter 6).

TSFC has two main features:

- It is a *structure-preserving compiler* in that it keeps intact the structure of algebraic operations (e.g., index sums, inner products), rather than committing to a specific implementation. This lets the lower-level compiler to explore the space of all possible transformations.

- As opposed to FFC, it supports the compilation of complicated forms making extensive use of tensor algebra. TSFC can efficiently identify repeated sub-expressions and assign them to temporary variables, thus drastically reducing the code generation time.

We observe that in Firedrake there is a neat separation of concerns:

- UFL is the mathematical language that allows expressing finite element problems.

- TSFC progressively abstracts away the domain knowledge and transforms the input into an AST. Some nodes in the ASTs are decorated to keep track of properties (e.g., linearity of an expression in a given set of symbols) exploitable for later optimization.

- COFFEE uses several rewrite operators for reducing the operation count of the operators returned by TSFC.

The COFFEE layer, design and optimization algorithms constitute one of the main contributions of this thesis. Chapters 4, 5 and 6 are entirely devoted to these topics.

Iteration over the Mesh

Finite element problems require the application of computational *kernels* over the discretized equation domain. In Firedrake, this is accomplished through *PyOP2* Markall et al. [2013], a domain specific language embedded in Python relying on just-in-time compilation and execution.

Typical kernels are the assembly operators presented in Section 2.1.4, usually the most expensive from a computational viewpoint. Other relevant examples are the application of boundary conditions as well as the interpolation and projection of fields. The fact that a kernel is a local operation – its application on an element of the mesh is independent of the application on any another element – suits naturally parallel execution.

This does not mean, however, that parallelizing a finite element computation is straightforward. As clarified in Section 2.2.2, a kernel can update a field either directly or indirectly. In the latter case, a subset of values, for instance the degrees of freedom at the boundary between two elements, may need be incremented by two different processes/threads, which requires non-trivial coordination.

PyOP2 supports the parallel application of kernels over unstructured meshes, a key requirement for FEMs. It also provides global data structures such as sparse matrices, which are essential when it comes to solve linear systems as 2.5.

This abstraction implements another separation of concerns. The kernels, which encode the numerics of an FEM, are produced at layers above PyOP2, usually in a completely automated fashion by Firedrake and the form compilers themselves. On the other hand, the parallel execution is completely masked by PyOP2.

The PyOP2 layer, and its relationship with the OP2 library Giles et al. [2011], are documented in mode detail in Section 2.2.2.

Unstructured Meshes

PyOP2 only works with sets and fixed-arity maps. Possible sets are topological entities such as cells or degrees of freedom. A map is an object describing the adjacency relationship between the elements of two distinct sets (e.g., a map from cells to degrees of freedom). PyOP2 has therefore no concept of what a mesh is or how it can be constructed. This task, in Firedrake, is carried out by another software module, PETSc's *DMPlex* **?**. The maps required by PyOP2 are then directly derived from DMPlex.

DMPlex is a data management abstraction representing unstructured mesh data through direct acyclic graphs. DMPlex relieves PyOP2 from the duty of handling two complex operations: partitioning for distributed parallel execution and reordering for efficient memory accesses. Moreover, the flexibility of the abstraction allows implementing techniques for communication-computation overlap, for instance via redundant computation along partition boundaries.

In this thesis, we will exploit the versatility of DMPlex for implementing automated loop fusion (Chapter 3).

Solving systems of linear equations

As we have seen, one of the key design principles characterizing Firedrake is exploiting successful, available software to carry out specific tasks of an FEM. This philosophy is also applied for what concerns the resolution of systems of linear equation, for which the *Portable, Extensible Toolkit for Scientific Computation* library **?**, or simply PETSc, is employed.

PETSc is entirely implemented in C, although its Python interface *petsc4py* makes it easily usable by frameworks such as Firedrake. It provides a wide range of algorithms for solving linear systems as well as a considerable number of options to drive the solvers. All algorithms are parallelized for distributed-memory execution through MPI. All these aspects probably make PETSc the most powerful tool when it comes to deal with expensive linear algebra operations.

Similarly to Firedrake, PETSc never attempts to reinvent science: many functionalities are implemented on top of existing libraries (e.g., BLAS) or offered via third-party implementations through suitable wrappers.

2.2.2 The PyOP2 and OP2 Libraries

PyOP2 is inspired from and shares many ideas with OP2[2] Giles et al. [2011], although it differs in a few yet significant ways. In this section, we first describe the main features of the abstraction and the implementation that are in common; we then conclude explaining how PyOP2 detaches itself from OP2.

Programming Model

OP2 offers abstractions for modeling an unstructured mesh, in terms of *sets* (e.g. vertices, edges), and mapping between them (e.g. mapping between edges and vertices) that express the mesh connectivity. An OP2 dataset associates data to each element of a mesh set (e.g. 3D coordinates).

OP2 programs, such as the one in Listing **??**, are expressed as sequences of parallel loops, each loop applying a computational kernel to every element in a mesh set. These *kernels* can access data associated to either the

[2]The name OP2 indicates that this is the second software engineering iteration of the OPlus library, or Oxford Parallel Library.

LISTING 4: Section of a toy OP2 program.

```
1  void kernel1 (double * x, double * tmp1, double * tmp2) {
2    *tmp1 += *x;
3    *tmp2 += *x;
4  }
5
6  // loop over edges
7  op_par_loop (edges, kernel1,
8               op_arg_dat (x, -1, OP_ID, OP_READ),
9               op_arg_dat (temp, 0, edges2vertices, OP_INC),
10              op_arg_dat (temp, 1, edges2vertices, OP_INC))
11
12 // loop over cells
13 op_par_loop (cells, kernel2,
14              op_arg_dat (temp, 0, cells2vertices, OP_INC),
15              op_arg_dat (temp, 1, cells2vertices, OP_INC),
16              op_arg_dat (temp, 2, cells2vertices, OP_INC),
17              op_arg_dat (res, -1, OP_ID, OP_READ))
18
19 // loop over edges
20 op_par_loop (edges, kernel3,
21              op_arg_dat (temp, 0, edges2vertices, OP_INC),
22              op_arg_dat (temp, 1, edges2vertices, OP_INC))
```

loop iteration set (direct access) or to other sets (indirect access) through mappings (or OP2 maps). As an example, a parallel loop iterates over all edges of the mesh and the kernel increments a dataset associated to vertices (i.e. each edge updates its two vertices). OP2 maps are implemented as arrays of indices.

The running example including three parallel loops. The first loop iterates over edges, as indicated by the first parameter of the *op_par_loop* function. In OP2, an iteration space (edges, in this case) is expressed as an op_set structured type, whose declaration is omitted in the example. The op_par_loop applies a kernel "kernel1" to every element in the indicated iteration space, i.e. to each edge. The kernel of this example reads data associated to an edge and increments the two adjacent vertices with the read value. This is indicated by the `OP_READ` and `OP_INC` modes in the access descriptors.

The access descriptors (or op_arg_dat) are used to indicate which datasets are being accessed when iterating on an OP2 set. When the access descriptor indicates OP_ID for a data array, then that data array is being accessed with the identity function. The second and third op_args in the example use indirections. Assuming that *temp* is a dataset associated to vertices, we need to tell OP2 what vertex data should be passed as second and third

parameters (i.e. what fields of *temp* are required for each invocation). The index array *edges2vertices* maps each edge index into the indices of its two incident vertices with the integer values of 0 and 1 indicating which of the two vertices should be considered.

Execution Model for Shared-Memory Parallelism

The execution order of parallel loop iterations does not influence the result. The best ordering scheme, therefore, can be chosen by the run-time implementation. For shared-memory-parallelism, in case of indirect increments/reductions (as in the example), OP2 guarantees data race avoidance by scheduling serially (in any order) iterations incrementing the same variable.

Data races are avoided by partitioning the vertices in the underlying mesh. Partitions that share an iteration set element (e.g. edge, cell) access shared vertex data and, therefore, are considered conflicting partitions. The partition conflict graph is eventually colored to enforce serialisation of increments to shared data. Partitions assigned the same color can be executed in parallel by different threads. On the other hand, partitions of different colors are scheduled serially, to prevent race conditions. The execution of elements inside a partition is serialized by scheduling them to a single thread.

Partitioning is performed on a per-loop basis. Each loop iteration set is partitioned by creating a dependency list between iterations using one of the maps used to access data in the loop. The built adjacency list can optionally be passed to a partitioning algorithm (METIS and PT-Scotch have been demonstrated for use in OP2) to maximize iteration locality according to their data access pattern. In other words, iterations accessing the same data will be more likely to be assigned to the same partition rather than independent iterations.

Execution Model for Distributed-Memory Parallelism

Distributed-memory parallelism is conceptually simpler than shared-memory parallelism. During the OP2 initialization phase, sets, maps, and datasets are partitioned and then distributed to different processes. For executing a parallel loop, MPI messages are usually exchanged to refresh the out-

of-date values along the partition boundaries. In overlap, each process computes its own portion of "local" iterations. Once both phases have finished, a process can correctly execute the remaining boundary iterations.

To implement this parallelization scheme, the iterations of each locally stored OP2 set are divided into four contiguous regions:

Core Iterations owned that can be processed without reading halo data.

Owned Iterations owned that can be processed only by reading halo data.

Exec halo Off-process iterations that are redundant executed because they indirectly increment owned iterations.

Non-exec halo Off-process iterations that only need be read to compute the exec halo iterations.

This situation is depicted in Figure **??**. Clearly, a good partitioning maximizes the ratio between the sizes of the core and non-core regions.

PyOP2 Features

PyOP2 distinguishes itself from OP2 in a number of features.

- An OP2 program is statically analyzed. The OP2 source-to-source compiler produces a legal C program, which is subsequently compiled and executed, possibly on a variety of different architectures. PyOP2, on the other hand, is entirely implemented in Python. Code is generated at run-time by inspecting the objects representing a program. The choice of Python makes PyOP2 easily composable with other abstractions. In the context of the Firedrake project, for example, PyOP2 relieves from the need for a fully-fledged compiler capable of translating arbitrary Python code into C (recall that UFL is embedded in Python). Rather, the analysis is for free as PyOP2 objects (e.g., sets, maps, parallel loops) are constructed as they are encountered during interpretation. A hierarchy of "software caches" minimizes the overhead of such an implementation by avoiding repeatedly generating code for kernels and parallel loops already "seen" during execution.

- Despite sharing relevant constructs (e.g., sets, maps), the PyOP2 domain specific language tends to be more compact and expressive than the OP2 counterpart. This is again a consequence of simplifying the analysis phase.
- PyOP2 supports global sparse matrices, basis linear algebra operations and mixed types (e.g., mixed sets), which are essential features for integration with a finite element framework. OP2 has none of these.
- OP2 completely handles distributed-memory parallelism, including partitioning and distribution of data structures as well as renumbering for increased data locality. PyOP2, as explained, relies on an external software module, DMPlex, for these features. The versatility of DMPlex will be fundamental for implementing and automating the performance optimization described in Chapter 3.

2.2.3 Stencil Languages

A stencil defines how to access a set of values associated with a grid point and its neighboring points. The word "stencil" often tacitly refers to the case in which the rule for accessing the neighboring points is expressed as an affine function. Many computational methods, typically based upon finite difference or close variants, can then be described by means of stencils operating on n-dimensional structured meshes. This definition can be generalized to include the relevant case of this thesis – that is, unstructured meshes – by relaxing the constraint about the affine nature of the access function. In particular:

Stencils for structured meshes Given an element $\boldsymbol{i}$ in the mesh, a stencil is a vector-valued function $f(\boldsymbol{i}) = [f_1(\boldsymbol{i}), f_2(\boldsymbol{i}), ..., f_n(\boldsymbol{i})]$ which retrieves the n elements that need be updated when accessing $\boldsymbol{i}$. A function f_j is affine and usually takes the form $f_j(\boldsymbol{i}) = \boldsymbol{i} * h + o$, where $h, o \in \mathbb{N}$ act as offsetting parameters.

Stencils for unstructured meshes Given an element $\boldsymbol{i}$ in the mesh and an affine access function g, a stencil is a vector-valued high-order function $f(\boldsymbol{i}, g) = [f_1(\boldsymbol{i}, g), f_2(\boldsymbol{i}, g), ..., f_n(\boldsymbol{i}, g)]$ which retrieves the n elements that need be updated when accessing $\boldsymbol{i}$. A function f_j is

non-affine and usually takes the form $f_j(\boldsymbol{i}, g) = g(\boldsymbol{i}) * h + o$, where $h, o \in \mathbb{N}$ act as offsetting parameters.

Stencil Languages for Unstructured Meshes

OP2 is an example of a language implementing stencils for unstructured mesh applications. The keyword "map" is used in the language to denote a non-affine stencil.

Yet another example of language for unstructured mesh stencils is Lizst [DeVito et al., 2011]. Similarly to OP2, Lizst supports multiple architectures, including distributed-memory execution via MPI and GPUs. The language is less flexible than that of OP2, though. Mesh elements such as vertices and cells are first-class citizens and fields can only be associated with mesh elements. This is from one hand helpful, because the stencils (i.e., the relationship between mesh elements) can be automatically inferred, assuming that the mesh topology does not change over time. On the other hand, it makes integration with a finite element framework difficult, or simply unnatural. Consider the case in which quadratic or higher-order basis functions on triangles are used. The degrees of freedom are associated with both vertices and edges. In OP2, this is naturally expressible by defining a map between cells and degrees of freedom, whereas in Lizst one needs managing two different fields (one for the degrees of freedom on the vertices and the other for those on the edges). A computation in Lizst is expressed through sequences of *for-comprehensions* over mesh elements. The for-comprehensions can be arbitrarily nested. One of the key prerequisites is that each field in a for-comprehension nest has a fixed state, one among read, write, or increment (for local reductions). This allows the compiler to automatically perform useful actions such as scheduling transformations (e.g., by rearranging iterations in a for-comprehensions nest) and placement of MPI calls. The Lizst compiler derives data dependency information automatically, while OP2 relies on access descriptors.

Stencil Languages for Structured Meshes

The field of domain specific languages for structured mesh computations has received a large number of contributions over the last decade.

SBLOCK **?** is a Python framework for defining structured stencils; the run-time supports automatically generates low-level code for multi-node many-core architectures. Mint **?** is a framework based on pragma directives targeting GPUs that has been used to accelerate a 3D earthquake simulation **?**. Other stencil languages relying on auto-tuning systems for increasing the performance of the generated code are **?**, **?**, **? ?**.

An interesting approach is adopted in Pochoir Tang et al. [2011], in which a compiler translates the high-level specification into cache-oblivious algorithms for multi-core CPUs. Interestingly, an attempt at integrating this system with a higher-level domain specific language for finite difference computations failed **?** due to constraints on the programming interface.

A stencil language explicitly aiming for generation of highly-efficient vector code is **?**.

A commonality characterizing all these works is that it is not clear to what extent stencil languages have been adopted in production code.

2.3 Compilers and Libraries for Loop Optimization

In this section, we review the state-of-the-art on compiler- and library-based approaches to loop optimization. This will provide the necessary background for the contributions presented in Chapter 3.

2.3.1 Loop Reordering Transformations

High-level transformations for optimization of loops in imperative languages date back to the sixties. The first studies focused on improving data locality and vectorization in loop nests. The main motivation was the observation that many programs spend a considerable fraction of time in executing these regions of code. A survey on the main results achieved between the sixties and the nineties was provided by **?**. Over the last twenty years, a great deal of effort has been invested in tools capable of automating these transformations as well as in developing techniques to increase their effectiveness. Excellent results have been achieved by many general-purpose compilers, which can now count on powerful loop transformation engines (e.g., the *Intel* and *Cray* compilers). Polyhedral compil-

ers, discussed in Section 2.3.3, have aimed for similar goals, although they have so far achieved controversial results.

A class of optimizations relevant for this thesis is the one based on the *reordering* of iterations of a loop nest, or a sequence of loop nests. Notable transformations belonging to this class are:

Interchange This transformation consists of exchanging the position of two loops in a nest. Possible objectives are exposing as innermost loop a vectorizable dimension or increasing data locality.

Reversal When "reversing" a loop, the order in which its iterations are executed is changed. For instance, instead of iterating from 0 to $n-1$ increasing the iteration variable by 1 at each loop iteration, the iteration order after reversal starts from $n-1$ and reaches 0 by decreasing the iteration variable. This transformation can enable other transformations or, under certain circumstances, eliminate the need for some temporary variables.

Skewing Loop skewing, often called "time skewing" to emphasize the fact that in many scientific computations the transformation changes the iteration order across the time dimension, aims to improve data locality in wave-front computations. In these particular loop nests, one or more arrays are updated at every loop iteration, and the updates propagate like a wave over the subsequent iterations (e.g., `A[i] = f(A[i-1])`). It can also be used to identify subsets of iterations that can be executed in parallel.

Fission Sometimes also referred to as "loop distribution", loop fission "splits" a a loop into a sequence of loops. The new loops have exactly the same iteration space of the original one, but only a subset of statements. This may increase the data locality in a loop in which a significant number of different datasets are accessed, at the price of increasing loop overhead.

For further information the reader is invited to consult **?**.

To this class also belong the two fundamental reordering transformations used in this thesis.

Tiling Also known as *blocking*, loop tiling is probably one of the most studied and powerful transformations. Its main aim is improving data locality. It achieves that by "chunking" the iteration space of a loop nest into partitions of a given shape. This requires major changes to the loop nest structure, as shown in Listing **??**; here, the "tiled" code executes, in sequence, "square tiles", thus increasing data locality. Depending on aspects such as the data dependency pattern and the control flow, implementing tiling may pose significant challenges. For example, ensuring the correctness of tiling requires a non-trivial analysis of the code if the loop nest includes a stencil. Among the first studies on loop tiling are **??**. More recent works on automation and scheduling strategies are **???**. The effects of different tile shapes have been analyzed in **??**.

Fusion A sequence of loops can be fused, or "merged", to improve data locality and reduce loop overhead. In the most simple case, all loops in a sequence have same iteration space and, given S_1 a statement in a first loop and S_2 a statement in a subsequence loop, S_2 does not modify any data read by S_1. In general, loops can have different bounds and different kind of dependencies may arise. A theoretical study on the complexity of loop fusion is **?**.

2.3.2 Composing Loop Tiling and Loop Fusion

Reordering transformations can be composed to maximize their effectiveness. A case of particular interest for this thesis is when fusion and tiling are combined. Here, a sequence of loops is first fused into a single loop; then, consecutive iterations are grouped together to form tiles. Such a transformation can exploit the data locality across consecutive loops, thus optimizing for memory bandwidth and latency. Unfortunately, automating or even just implementing such an optimization can be cumbersome depending on what kind of loops need be supported.

There is a large body of research on the composition of fusion and tiling for structured stencil codes, for example Bondhugula et al. [2014], **?**], **?** as well as most articles concerning polyhedral compilation (reviewed in Section 2.3.3), such as Bondhugula et al. [2008]. When applied to computational methods for solving partial differential equations, this trans-

formation has often been called *time tiling*. In these codes, a sequence of loops over the spatial dimensions is iteratively executed within a time loop. Time tiling fuses the spatial loops and builds tiles spanning the time dimension.

To a much smaller extent, fusion and tiling have been applied in the context of unstructured stencil codes. The composition of these two transformations takes the name *sparse tiling*. What makes it difficult to apply sparse tiling is the data dependency pattern induced by the stencil, which cannot be analyzed statically. Early studies focused on "ad-hoc techniques" for individual benchmarks relying on so called *inspector/executor strategies*. These will be extensively reviewed in Section 3.4. In essence, this thesis advances the state-of-the-art by introducing:

- a technique to fuse and tile a sequence of loops including unstructured stencils, or "irregular loops". The sequence and the loop dependencies can be arbitrary as long as they are expressible through the so called *loop chain abstraction* Krieger et al. [2013]. This contribution was a joint effort with the authors in Strout et al. [2014].

- a system that fully automates this technique and enables execution on distributed-memory architectures.

Independently of the stencil structure, there are two basic approaches for composing fusion and tiling:

Split tiling The loops are fused by computing tiles and dependencies between tiles. The tiles are then scheduled such that all tile-to-tile dependencies are satisfied. In the context of structured stencil codes, over the last decade a great deal of effort has been invested in finding algorithms to compute effective split tiling schemes. These schemes take their name from the resulting shape of the tiles into which an iteration space is partitioned, such as *hexagonal tiling* or *diamond tiling*. Split tiling schemes usually differ in the trade-off between achieved parallelism and data locality.

Overlapped tiling The loops are fused by computing overlapping tiles; that is, tiles that share a subset of iterations. The shared iterations are usually "owned" by a specific tile and are executed redundantly by

a set of tiles, which store intermediate values into "ghost" regions of memory. This approach allows all tiles to execute in parallel without the need for any form of synchronization, although at the price of redundant computation.

In this thesis, we employ a mixed split-overlapped scheme for effective tiling of unstructured stencil codes on distributed-memory architectures. A survey on different tiling techniques is provided by **?**.

2.3.3 Automation via Static Analysis

There exist two different approaches to the automation of loop reordering transformations, and both of them rely on static analysis of source code:

Graph-based Representation General-purpose compilers analyze the source code and produce an intermediate representation (IR) usually based on a graph-like data structure (e.g., SSA in LLVM). Algorithms and cost models are used to assert the legality of a transformation and its potential impact on performance. Sometimes the cost models are made visible to the user by suitable compiler reports (e.g., detailed vectorization reports can be requested to the Intel compiler). Users have some form of control over the optimization process. Pragma directives can be used to explicitly choose how to optimize a loop nest (e.g., to set a specific unroll factor or to enforce vectorization when the compiler erroneously thinks it is unprofitable), while compiler parameters can be tuned to change global optimization heuristics.

Polyhedral Model Several research compilers, and more recently a fork of the LLVM compiler itself through a module called Polly Grosser et al. [2012], apply reordering transformations based on geometric representations of loop nests. To model a loop nest as a *polyhedron*, two conditions must be satisfied: (i) loop bounds as well as array indices are expressions affine in the loop indices and (ii) pointer aliasing known at compile-time. These conditions are often necessary in the case of graph-based IRs too, although in a more relaxed fashion (e.g., the Intel compiler can vectorize, to some extent, non-affine loop nests). Polyhedral compilers target parallelism and data locality by composing *scheduling functions*. A schedule defines the

order in which the iterations of a loop nest are executed; a scheduling function can be applied to change the original order. Once a polyhedron is available, a scheduling function can be constructed and cost models applied to assess its potential.

To apply a reordering transformation, a compiler – irrespective of the IR employed – first needs to build a model of the data dependencies pattern. If this model can be built and the optimization is determined to be legal and profitable – these steps require a sophisticated analysis of the code region – then the transformation is applied by suitable manipulation of the IR.

If, from one hand, general-purpose compilers using graph-based IRs have now reached an impressive level of sophistication (our experience with the Cray and Intel auto-vectorization systems is remarkably positive), there is still quite a lot of debate on the effectiveness of polyhedral compilers. Even for a state-of-the-art polyhedral compiler like PLUTO [Bondhugula et al., 2008] it is difficult to find realistic applications in which significant performance improvements have been achieved. We will review the main causes of this weakness in Section 3.3, with emphasis on the applicability of fusion and tiling to codes arising in scientific simulations.

In this thesis we focus on irregular codes – that is, where the loops have unstructured stencils – as those arising in finite element and finite volume methods. Unstructured stencils make loop nests non-affine, thus precluding the adoption of polyhedral compilation. Recently, there has been some effort on extending the polyhedral model to non-affine loops **?**, but we hardly believe this model will ever be capable of handling the complexity inherent in the real-world codes we are interested in.

2.3.4 Automation via Dynamic Analysis

If a stencil is unstructured, the memory access pattern is characterized by indirect accesses that can only be resolved at execution time. In such a case, loop reordering transformations can be enabled through an inspector/executor strategy, as originally proposed in Salz et al. [1991]. Informally, an inspector is an additional piece of code that captures the data dependency pattern of an irregular loop nest into a suitable data structure. An executor is semantically equivalent to the original loop nest, but

it changes the iteration order by exploiting the information produced by the inspector. Examples of reordering transformations through inspector/executor schemes were provided by **?**.

To automate a loop reordering transformation in presence of unstructured stencils, a mixed compiler/run-time system approach is required, with the compiler searching for loops amenable to dynamic analysis and replacing them with suitable inspector/executor schemes.

2.4 Domain Specific Optimization

By targeting affine loop nests, polyhedral compilers are theoretically applicable in a wide range of programs, including computational methods for approximating the solution of partial differential equations, linear algebra routines, and image processing kernels. From our perspective, the class of affine loop nest is still quite generic however. As observed in **?**, by restricting the attention to narrower fields or classes of programs, it is relatively simple to identify optimization opportunities that, due to their domain-specific nature (e.g., they rely on some mathematical properties), will be missed by any general-purpose of polyhedral compilers. Optimization systems specialized for specific classes of codes can therefore achieve significant performance gains over more general compilers. This observation is one of the cornerstones of this thesis, as one can appreciate from the contributions in Chapters 3, 4 and 5. In this section, we review some of the most important domain optimization systems that somehow inspired our work.

2.4.1 Tensor Contraction Engine

The Tensor Contraction Engine (TCE) is a successful story of how the mathematical structure of expensive computations in the field of quantum chemistry can be turned into powerful optimizations **?**. These codes need execute large sequences of tensor contractions, or generalized matrix multiplications, which can easily result in teraflops of computation and terabytes of data for simultaneously storing huge dense matrices. The TCE provides a domain-specific language to express formulae in a mathematical style. The mathematical specification is then transformed into

low-level code undergoing several optimization steps. Transformations for reducing the operation count **??**, for finding the best trade-off between redundant computation and data locality **??**, and for more generic low-level optimization **?** are available. Most of these optimizations exploit the mathematical structure inherent in tensor contractions.

In Chapter 4 we use a similar approach for optimizing the operation count of finite element operators – we exploit the mathematical property that these operators are linear in test and trial functions to identify effective factorizations.

2.4.2 Halide

? recently introduced Halide, a high level language for expressing image processing kernels. The run-time optimization system, which exploits auto-tuning to explore a large space of transformations, has been demonstrated to achieve a performance at least comparable to that of hand-written (and hand-optimized) code, and in many cases to outperform them. Halide is another successful story in the context of domain specific languages and domain specific optimization systems, since it is currently employed for development by several companies[3].

Halide allows users to define image processing pipelines. These are sequences of interconnected stages, each stage applying a numerical kernel – usually a structured stencil or reductions – to their input. Numerical kernels are pure functions applied to a 2D domain representing the image being processed. In realistic cases, an image processing pipeline can be quite complex. For example, in **?** it is shown a pipeline with 99 stages. What makes Halide powerful from the viewpoint of optimization is the fact that the schedules are decoupled from the numerical kernels. A schedule describes aspects like the iteration ordering and the trade-off between temporary values and redundant computation. These optimizations are fundamental in image processing pipelines and, as such, are treated as first-class citizens by Halide. Different schedules can be explored automatically or provided as user input.

[3] At least Google and Adobe have declared that some of their groups (more than 30 researchers and developers in total) are actively using Halide.

2.4.3 Spiral

Spiral is one of the pioneering projects on automated code generation from a high level specification of a mathematical problem **?**. The domain of interest is the one of digital signal processing (DSP). Spiral generates highly optimized DSP algorithms, such as the discrete Fourier transform, and autonomously tunes them for the underlying platform. To achieve that, the mathematical specification of a DSP algorithm is first transformed according to a set of pre-established rewrite rules. The resulting formulae are translated into an intermediate language, which enables a set of optimizations, including explicit vectorization and parallelization. Finally, low level code is produced, compiled, executed and timed. The last phase eventually provides feedback to the system so that increasingly optimized implementations can be generated.

Spiral provides several examples of how the mathematical knowledge can be turned into powerful optimization algorithms. The rewrite rules system itself – as mentioned above, essential for simplifying otherwise extremely complicated formulae – is one such example. Yet another example is the framework used by the Spiral compiler to identify loop fusion opportunities **?**. The general loop fusion problem is NP-complete **?**; the domain knowledge, however, allows Spiral to apply effective search algorithms that overcome the obvious limitations of any possible lower level compiler.

2.4.4 Small-scale Linear Algebra

In several fields, such as graphics, media processing and scientific computing, many operations can be cast as small-scale linear algebra operations. By small-scale we mean that the size of some of the involved tensors can be as small as a few units, and only occasionally exceeds a few hundreds elements. Despite the small size, it is important to optimize these operations because they may be applied iteratively (e.g., in a time-stepping loop), thus accounting for a significant fraction of the overall execution time.

Libraries for linear algebra are tuned for large-scale problems and they become inefficient when tensors are small. Novel approaches, mostly centred on auto-tuning, have been developed. In the domain of scientific

computing, it is worth mentioning the technique employed by the finite element code *nek5000* [**?**] to optimize small matrix multiplications [Shin et al., 2010]. A set of highly-optimized routines, each routine specialized for a particular size of the input problem, are generated and tuned for the underlying platform. At run-time, a dispatcher function picks one such routines given the size of the input matrix multiplication. A higher level approach has recently been presented in Spampinato and Püschel [2014], where the *LGen* language is used to write composite linear algebra operations. A set of rewrite rules and a transformation system deeply inspired by Spiral are used for optimization.

The field of small-scale linear algebra optimization systems is interesting because some techniques could be used for low-level optimization of finite element local assembly, a topic treated by this thesis in Chapter 5.

2.5 On the Terminology Adopted

Throughout the thesis we employ a standard terminology, very close to the one used in reference textbooks such as **?**. We here review a set of relevant keywords. This will especially be useful when discussing the performance achieved by the proposed optimizations.

Compilers

General-purpose compiler With this term we generically refer to any open-source or commercial compilers capable of translating low level source code (e.g., Fortran, C, C++) into machine code. Examples are the GNU (*gcc*), Intel (*icc*), Cray and LLVM compilers. With "general-purpose" we aim to distinguish them from other (higher level) compilers, such as polyhedral and domain specific language compilers, which we also need to refer to when discussing code transformations.

Autovectorization Vectorization is a well-known paradigm that generalizes computation on scalars to computation on vectors – that is, an array of contiguous elements. A single instruction, multiple data (SIMD) computation is one that employs vectorization to carry out a sequence of instructions. SIMD architectures, which are nowadays

ubiquitous, emit vector code in two circumstances: (i) sections of a program are explicitly vectorized (e.g., through high level libraries, intrinsics instructions, or assembly code); (ii) a compiler transforms scalar code into vector code. The latter case is often referred to as auto-vectorization, since SIMD instructions are generated without user intervention. When possible and demonstrated to be effective, auto-vectorization should be preferred over explicit vectorization for portability reasons. Auto-vectorization is traditionally applied to (inner) loops, although block vectorization **?** is also supported by more powerful compilers (e.g., Intel's).

Access function An access function specifies how the elements of an array are accessed. Usually, these are functions of one or more loop indices. Access functions can be constant, affine or non-affine, as already shown in Section 2.2.3.

Local and global reductions A reduction is a commutative and associative operation that is applied to a set of values to produce a scalar. For instance, the sum of a set of numbers is a reduction. In mesh-based computations, it is useful to distinguish between local and global reductions. A reduction is local if only applied to a (typically small) subset of mesh elements. A reduction is global if applied to an entire set of elements (e.g., a field associated with a set of degrees of freedom), thus introducing a global synchronization point in the computation.

Communication A communication indicates a generic form of interaction between two or more entities. The most obvious case is when two processes on two different cores communicate explicitly via message passing; if the cores are on the same node the communication occurs via memory, whereas if they are on different nodes both the network and the memory are needed. However, the term can also be used in more general scenarios. We can say, for instance, that two tiles in a blocked iteration space communicate if their execution needs be synchronized. Intuitive terms like *communication-avoiding* or *communication-computation overlap* are often used to classify optimizations that aim to minimize communication.

Performance and Cost Models

Memory pressure This is often used as an umbrella term to emphasize the fact that ore or more levels of the memory hierarchy (e.g., RAM, caches, registers) are stressed by a relatively large number of load/-store instructions. A high memory pressure is often responsible for performance degradation because it causes memory-boundedness.

Memory- and CPU-boundedness At high level, a section of code can be either CPU-bound or memory-bound. In the former case, the performance achieved is limited by the operation throughput of the CPU; in the latter case, the memory bandwidth or the memory latency are the limiting factors. The loop reordering transformations reviewed in Section 2.3 tackle memory-boundedness; for example, both tiling and fusion aim to maximize the cache hit ratio, thus reducing latency and memory pressure. Many domain specific optimizations, as discussed in Section 2.4, instead target CPU-boundedness; for example, the Tensor Contraction Engine and Spiral manipulate mathematical formulae to reduce the operation count of the resulting kernels.

Operational intensity and Roofline Model This parameter defines the ratio of total operations to total data movement (bytes) between the DRAM and the cache hierarchy for a given section of code. The operational intensity, which "*predicts the DRAM bandwidth needed by a kernel on a particular computer*, is useful to derive *roofline plots* **?**. This tool is particularly helpful to study the computational behaviour of a program, since it provides an insightful mean to understand what the performance bottleneck is – if any – and, therefore, what kind of optimization is most useful.

Arithmetic intensity Sometimes, the term *arithmetic intensity* is used in place of *operational intensity*. The differences are that only the fraction of arithmetic operations emitted, instead of all operations, is considered and that the total data movement is to be interpreted as between the CPU and the last level of cache.

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 2.3.3 we have explained that loop fusion and loop tiling may 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 2.3.4, 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 **??** 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 general technique, applicable to arbitrary computations on unstructured meshes.
- Automation is more than a desired feature because application specialists tend to avoid complicated optimizations that harm source code comprehensibility. We therefore aim for a fully-automated framework, based upon a mixed compiler/library approach.
- Very few studies have tackled fusion when loops are interleaved by routines for message passing. We are aware of none for the case in which the memory accesses pattern is irregular, as in unstructured mesh applications. We describe and implement a technique that solves this problem. This is a fundamental contribution because most scientific simulations run on multi-node architectures.

3.2 Open Problems, Questions, Hypotheses

Loop fusion and loop tiling have been widely studied in the literature. However, it is not clear to what extent they are actually employed 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 these transformations to "simple" memory-bound loop nests can result in considerable speed-ups. Some of the most notable examples are stencil codes arising in finite difference methods [Zumbusch, 2013, Holewin-

ski et al., 2012, Bondhugula et al., 2014], linear algebra routines [Buttari et al., 2008, 2009], and image processing kernels [Ragan-Kelley et al., 2013]. 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 wider audiences?

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 co-exists with message passing for distributed-memory parallelism.

Automation, but no legacy code Sparse tiling is an "extreme optimization". It requires a great deal of effort to be implemented because it imposes a complete restructuring of the core of the application. Similarly to many other low level transformations, it also tends to render 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 interested 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 observations and requirements, we decompose our problem into two tasks:

1. a library for writing inspector/executor schemes (Salz et al. [1991]) to enable sparse tiling in arbitrary computations on unstructured meshes;

2. integration of the library with a multilayer framework based on DSLs and runtime code generation.

Before addressing these two tasks, respectively in Sections 3.5-3.7 and Section 3.8, we elaborate on the theoretical and technical challenges that arise when applying loop fusion and tiling (Section 3.3), and review the related work (Section 3.4).

3.3 Applying Loop Fusion and Tiling is More Difficult than Commonly Thought

We show in Listing 5 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 neglected, or at least treated with scarce emphasis, in the relevant literature.

Theoretical questions We first wonder about the effectiveness of fusion and tiling in unstructured mesh applications.

LISTING 5: 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 the invocation of a kernel: first, the memory indirections are resolved; the kernel, which receives data that is now contiguous in memory (this hopefully increases the chances of vectorisation), is then invoked; finally, the computed values are "scattered" back to memory. Distributed-memory parallelism is achieved through MPI; messages are exchanged between processes (`MPI_Comm (...)`) 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 spars tiling on data locality will be marginal. Before even thinking about aggressive optimizations such as fusion and tilling, 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).

Loop 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 possible instances of the scheduling function of a given loop nest (see Section 2.3.3). We could not find studies comparing the performance of the two approaches. This seems to denote a lack of communication between communities that have tackled similar problems for years.

Practical issues Several studies on fusion and tiling for structured mesh applications have addressed key problems such as automation (e.g., polyhedral compilers), time-loop tiling (i.e., time skewing), exotic tile shapes for communication minimization (e.g., diamond tiling). However, the following issues were rarely given the right importance.

Unstructured meshes Although ad-hoc inspector-executor strategies for some proxy applications had previously been developed, a general technique to fuse and tile arbitrary computations on unstructured meshes has been missing until this thesis[1]. As already explained, the main problem with unstructured meshes is the presence of indirect memory accesses, which essentially inhibit the static analysis needed by any loop transformation.

Time tiling and MPI We reiterate the fact that real-world computations almost always run on distributed-memory architectures. This is evident in Listing 5, 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.

[1] We reinforce once more that the generalized sparse tiling algorithm is the result of a joint collaboration among the authors of Strout et al. [2014].

Time tiling and extra code The `Comp(...)` function in Listing 5 denotes the possibility that additional computation is performed between consecutive sweeps. This could be, for instance, checkpointing or I/O. Also, conditional execution of loops (e.g., through `if-then-else`) breaks time tiling.

Legacy code is usually impenetrable Loop transformation opportunities are often hidden in existing scientific codes. As explained in Strout [2013], common problems are: 1) potentially fusible or tilable 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.

All these issues and questions will be progressively addressed in the upcoming sections.

3.4 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 Krieger et al. [2013]. 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 3.5.

Inspector/Executor and Sparse Tiling

The loop chain abstraction provides sufficient information to create an inspector/executor scheme for an arbitrary unstructured mesh application. Inspector/executor strategies were first formalized by Salz et al. [1991]. They have been used to fuse and tile loops for exploiting data reuse and providing parallelism in several studies [Douglas et al., 2000, Strout et al., 2002, Demmel et al., 2008, Krieger and Strout, 2012].

Sparse tiling, which we introduced in Section 2.3.4, is the most notable technique based upon inspection/execution. The term was coined by Strout et al. [2002, 2004] in the context of the Gauss-Seidel algorithm and in Strout et al. [2003] in the context of the moldyn benchmark. However, the technique was initially proposed by Douglas et al. [2000] 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. Adams and Demmel [1999] also developed an algorithm very similar to sparse tiling, to parallelize Gauss-Seidel computations. The main difference between Strout et al. [2002, 2004] and Douglas et al. [2000] 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 Ravishankar et al. [2012] 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 2.3.3), have been applied to structured mesh benchmarks or proxy applications. Notable examples are Bondhugula et al. [2008], Grosser et al. [2012], Klöckner [2014]. 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 was 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 Bassetti et al. [1998]. Overlapped tilling (see Section 2.3.2) exploits the same principle to reduce communication at the expense of performing redundant computation along the boundary Zhou et al. [2012]. Several studies centered on overlapped tiling within single regular loop nests (mostly stencil-based computations), for example Meng and Skadron [2009], Krishnamoorthy et al. [2007], Chen et al. [2002]. Techniques known as "communication avoiding" [Demmel et al., 2008, Mohiyuddin et al., 2009] 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].

3.5 Generalized Inspector/Executor Schemes

In this section we explain how to build inspector/executor schemes for arbitrary sequences of loops.

3.5.1 Relationship between Loop Chain and Inspector

The *loop chain* is an abstraction introduced in Krieger et al. [2013]. 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 3.3 that the indirect memory accesses inhibit static optimization for data locality in unstructured mesh applications. The idea to work around this issue is to exploit the data dependence information carried by a loop chain to replace compile-time with run-time analysis. The source code must be modified adding a description of the loop chain and the inspector, as well as by replacing a sequence of loops with a new 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 somewhat provided. One possibility is that application specialists manually change their

programs by constructing the loop chain and the inspector/executor scheme through library calls. As already observed, this is clearly challenging. Another possibility is that the loop chain is constructed at a higher layer of abstraction, in which case the optimization process is fully automated. The tools we have built enable both approaches.

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

3.5.2 The Loop Chain Abstraction

In Krieger et al. [2013], a loop chain $\mathbb{L}$ is defined as follows:

- $\mathbb{L}$ consists of n loops, $L_0, L_1, ..., L_{n-1}$. 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, $D_0, D_1, ..., D_{m-1}$. Each loop accesses (reads from, writes to) a certain subset of these data spaces. An access can be either direct (e.g., `A[i]`) or indirect (e.g., `A[map(i)]`).

- $R_{L_l \to D_d}(\vec{i})$ and $W_{L_l \to D_d}(\vec{i})$, where the R and W access relations are defined over for each data space $D_d \in D$. 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)] = ...` in loop L_j, the access relation $B_{L_j \to A}(\vec{i})$ will be available).

3.5.3 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 2.2.2), the definition of a loop chain $\mathbb{L}$ is changed as follows:

- $\mathbb{L}$ consists of n loops, $L_0, L_1, ..., L_{n-1}$. 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.

- $\mathbb{S}$ is a set of disjoint m sets, $S_0, S_1, ..., S_{m-1}$. Possible sets are the cells in the mesh or the degrees of freedom associated to a certain function. Sets are used to define both iteration spaces and datasets. In the former case, a set element is an iteration; in the latter case, a set element is logically associated with a data value (not necessarily a scalar).

  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^{owned}) and exec (S^{exec}). S^{owned} are iterations that belong to P which will be redundantly executed by some other processes; S^{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 $S = (\texttt{vertices}, C, B, N)$ 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, $M_0, M_1, ..., M_{k-1}$. A map of arity a is a vector-valued function $M : S_i \rightarrow S_j^0 \times S_j^1 \times ... \times S_j^{a-1}$ 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, $D_0, D_1, ..., D_{d-1}$. A descriptor D is a 2-tuple $D = <M,$ `mode`$>$. M is either a map from S_j to some other sets or the special placeholder $\perp$, which indicates that the access is direct to some data associated with S_j. `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 $|\mathbb{S}| << |\mathbb{D}|$.

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.

3.6 Loop Chain and Inspection Examples

Using the example in Listing 6 – a plain C implementation of the unstructured mesh OP2 program illustrated in Listing 4 – we describe the actions performed by a sparse tiling inspector. The inspector takes as input a loop chain, as illustrated in Listing 7. 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 7.

Overview

The inspector starts with partitioning the iteration space of a *seed loop*, for example L_0. Partitions are used to initialize tiles: the iterations of L_0

LISTING 6: 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 }
```

falling in P_i – or, in other words, the edges in partition P_i – are assigned to the 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, $[T_0,\ T_1,\ T_2,\ T_3]$.

In the later stages of the inspection, T_i is populated with iterations from L_1 and 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 honored. In the next two sections, we use the running example to illustrate how iterations are assigned to tiles to enable shared-memory and distributed-memory parallelism.

The result of the inspection is eventually passed to the executor. The inspection carries sufficient information for a parallel computation of different tiles. A given tile T_i is always executed by a single thread/process "atomically"; that is, without ever communicating with other threads/processes. When executing T_i, first all iterations from L_0 are executed, then all iterations from L_1 and finally those from L_2.

LISTING 7: 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;
```

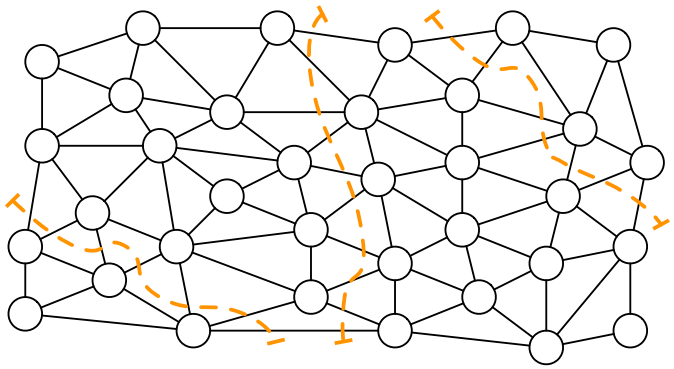


Figure 3.1: Partitioning of the seed loop.

Inspection for Shared-Memory Parallelism

Similarly to OP2, to achieve shared-memory parallelism we use 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 during parallel execution. 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 (see Listing 6) to a tile. Consider

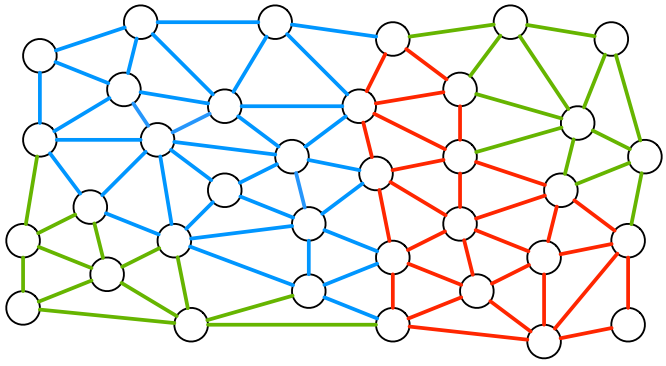


Figure 3.2: Tiling L_0.

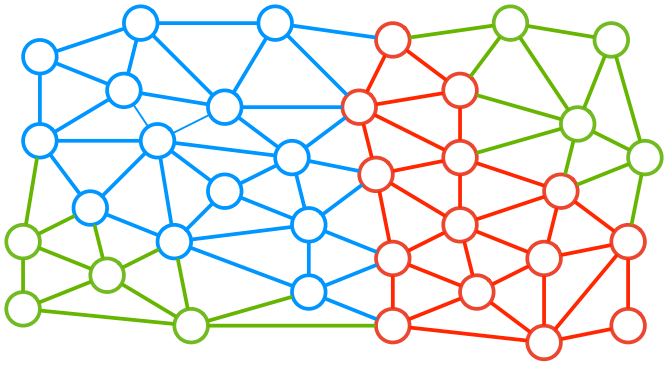


Figure 3.3: Projection of L_0 to L_1.

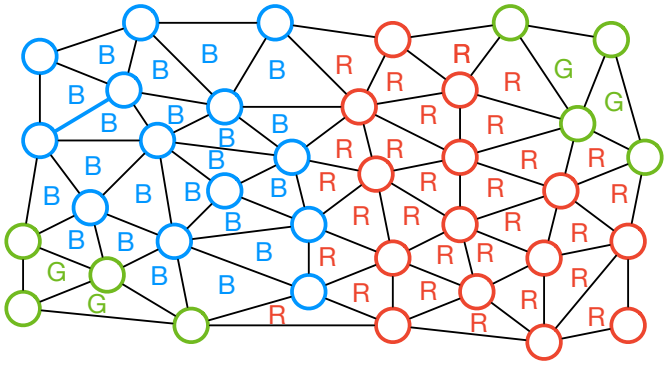


Figure 3.4: Tiling L_1.

the vertex v_0 in Figure 3.3. v_0 has 7 incident edges, 2 of which belong to T_0^G, while the remaining 5 to T_1^B. Since we established that $G \prec B$, v_0 can only be read after T_1^B has finished executing the iterations from L_0 (i.e., the 5 incident blue edges). We express this condition by setting $\phi(v_0) = T_1^B$. Observe that we can compute ϕ by iterating over V and, for each vertex, applying the maximum function (MAX) to the color of the adjacent edges.

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

tiles before all iterations from $[L_0, L_1, L_2]$ in the blue tiles". 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 ϕ to schedule the edges in L_2. Figure 3.5 shows the result of this last phase.

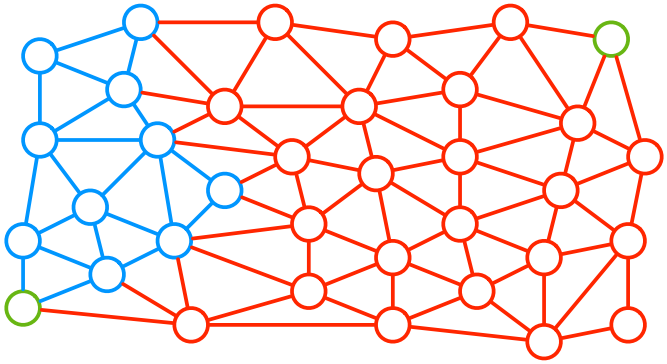


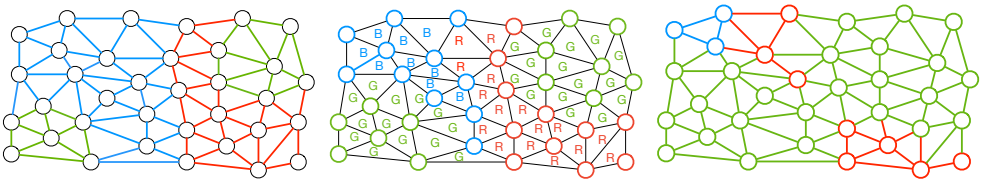
Figure 3.5: Tiling L_2.

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 different execution order – blue (B), red (R), and green (G) – and initial partitioning. We recall that the total ordering of colors is indeed arbitrary. 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". Race conditions during the execution of iterations belonging to L_2 are now theoretically possible. We will solve this problem in Section 3.7.2.

Inspection for 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 List-



(a) After tiling L_0 (b) After tiling L_1 (c) After tiling L_2

Figure 3.6: Tiling the program in Listing 6 for shared memory parallelism can lead to conflicts. Here, the two green tiles eventually become adjacent, exposing potential race conditions.

ing 5, MPI calls now separate the execution of two consecutive loops in the chain. This means that our inspector/executor scheme will have to take extra care of data dependencies along the mesh partition boundaries.

From Section 3.5.3 we know that all sets are logically split into three regions, *core*, *boundary*, and *non-exec*. The boundary region includes all iterations that cannot be executed until the MPI communications have successfully been accomplished. With this in mind, we take again L_0 as seed loop and make the following adjustments to the procedure illustrated in the previous section:

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 a group of tiles . We can assign colors increasingly: T_i is given color i. This allows us to uniquely identify a tile's color with its index, i. As long as tiles with contiguous ID are also physically contiguous in the mesh, this assignment retains spatial locality when "jumping" from executing T_i to T_{i+1}.

2. We replicate for the boundary region of L_0 what we just did for the core. By neatly separating the core and boundary regions, we prevent tiles from crossing the two regions by construction. Further, all tiles within boundary have greater color than those in core, which poses 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 in later sections.

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. The whole boundary region "expands" over core, and so does T_{ne} over boundary (see Figure **??**).

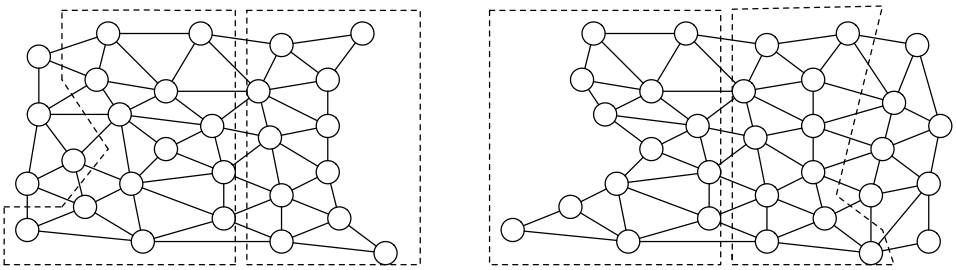


Figure 3.7: Seed partitioning for distributed memory execution.

On each process, the executor starts with triggering the MPI communications required for the execution of (all tiles over) the boundary regions; it proceeds to scheduling the core tiles, thus overlapping communication with computation; eventually, as soon as all core tiles have been executed and the MPI communications been accomplished, it computes the boundary tiles.

Finally, we observe that:

Correctness This inspector/executor scheme relies on redundant computation along the boundaries of mesh partitions. The depth of the boundary region – a parameter of the loop chain, as explained in Section 3.5.3 – grows proportionally with the number of loops that are fused. 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, although it poses a big challenge on software engineering.

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

3.7 Formalization

3.7.1 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 3.5, provides enough information to construct an inspector which analyzes all of the dependencies in a computation and builds a legal sparse tiling. We recall that one of the main assumptions in a loop chain is that each loop is fully parallel or, equivalently, that there are no loop carried dependencies.

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

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

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

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

While the output dependencies between loops L_x and L_y are:

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

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

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

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

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

As seen in the example in Section 19, 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.

3.7.2 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
ϕ_S	A projection $\phi_S : S \to \mathbb{T}$
Φ	The set of all available projections
σ_i	The tiling function $\sigma_i : S_i \to \mathbb{T}$ 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 is that $L_{seed} = L_0$.

In the special case in which there is no need to distinguish between core and boundary tiles (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 ($L_0 \prec ... \prec L_{seed} \prec ...$), a mechanism for constructing tiles in the reverse direction ("backwards"), from L_{seed} towards L_0, is necessary. In Strout et al. [2014], 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 $\mathbb{L} = [L_0,\ L_1,\ ...,\ L_{n-1}]$, a tile size ts
Output: A set of tiles $\mathbb{T}$, populated with iterations from $\mathbb{L}$

```
   // Initialization
1  seed ← 0;
2  Φ ← ∅;
3  C ←⊥;

   // Creation of tiles
4  σ_seed, 𝕋 ← partition(S_seed, ts);
5  seed_map ← find_map(S_seed, 𝕃);
6  conflicts ← false;

   // Schedule loops to tiles
7  do
8      color(𝕋, seed_map);

9      for i = 1 to n − 1 do
10         project(L_{i−1}, σ_{i−1}, Φ, C);
11         σ_i ← tile(L_i, Φ);
12         assign(σ_i, 𝕋);
13     end for

14     if found_conflicts(C) then
15         conflicts ← true;
16         add_fake_connection(seed_map, C);
17     end if
18 while conflicts;

   // Inspection successful, create local maps and return
19 create_local_maps(𝕋);
20 return 𝕋
```

this distinction[2].

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

[2]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.

We then create $m + k + 1$ tiles, $\mathbb{T} = \{T_0^c, ..., T_{m-1}^c, T_m^b, ..., T_{m+k-1}^b, T_{m+k}^n\}$, 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 $\phi_S : S \to \mathbb{T}$. Projections are computed and/or updated after tiling a loop. Initially, the set Φ of all projections is empty. Starting with the seed tiling $\sigma_{seed} : S_{seed} \to \mathbb{T}$, we iteratively update Φ and build tiling functions for all other loops $[L_1, L_2, ..., L_{n-1}]$.

ALGORITHM 2: Projection of a tiled loop

Input: A loop L_i, a tiling function σ_i, the projections set Φ, the conflicts matrix C
Result: Update Φ and C

```
1  foreach D in L_i.descriptors do
2      if D.mode == r then
3          skip;
4      end if
5      if D.map ==⊥ then
6          Φ = Φ ∪ σ_i;
7      else
8          inverse_map ← map_invert(D.map);
9          S_j, S_i, values, offsets ← inverse_map;
10         φ_{S_j} ←⊥;
11         for e = 0 to S_j.size do
12             for k = offsets[e] to offsets[e+1] do
13                 adjacent_tile = 𝕋[values[k]];
14                 max_color ← MAX(φ_{S_j}[e].color, adjacent_tile.color);
15                 if max_color ≠ φ_{S_j}[e].color then
16                     φ_{S_j}[e] ← adjacent_tile;
17                 end if
18             end for
19         end for
20         update(C, 𝕋, φ_{S_j});
21         Φ = Φ ∪ φ_{S_j};
22     end if
23 end foreach
```

Projecting tiled sets Algorithm 2 takes as input σ_{i-1} and (the descriptors of) L_{i-1} to update Φ. Further, the conflicts matrix $C \in \mathbb{N}^{m \times m}$, indicating 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 ϕ_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 $M : S_i \rightarrow S_j^0 \times S_j^1 \times ... \times S_j^{a-1}$. To compute ϕ_{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 Φ
Output: The tiling function σ_i

```
1  σ_i ← ⊥;
2  foreach D in L_i.descriptors do
3      if D.map == ⊥ then
4          σ_i ← Φ[S_i];
5      else
6          arity ← D.map.arity;
7          φ_S ← Φ[D.map.S_j];
8          for e = 0 to S_i.size do
9              σ_i[e] ← T_⊥ ;
10             for k = 0 to arity do
11                 adjacent_tile ← φ_S[D.map.values[e*arity + k]];
12                 max_color ← MAX(σ_i[e].color, adjacent_tile.color);
13                 if max_color ≠ σ_i[e].color then
14                     σ_i[e] ← adjacent_tile;
15                 end if
16             end for
17         end for
18     end if
19 end foreach
20 return σ_i
```

Scheduling loops Using Φ, we compute σ_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 Φ to schedule iterations correctly. Finally, σ_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.

3.7.3 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  T^c, T^b ← group_tiles_byregion(T);
2  start_MPI_communications();
3  foreach available color c do
4  |   foreach T ∈ T^c s.t. T.color == c do
5  |   |   execute_tile(T);
6  |   end foreach
7  end foreach
8  end_MPI_communications();
9  foreach available color c do
10 |   foreach T ∈ T^b s.t. T.color == c do
11 |   |   execute_tile(T);
12 |   end foreach
13 end foreach
```

communications; execution of boundary tiles.

As seen in the example in Section 19 and based on the explanation in Section 3.7.2, 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 parameter 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 $n > depth$ the loop chain must first be split into smaller sequences of loops, to be individually inspected and executed.

3.7.4 Computational Complexity of Inspection

Let N be the maximum size of a set in $\mathbb{L} = [L_0, L_1, ..., L_{n-1}]$ 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.

3.8 Implementation

The implementation of generalized sparse tiling is distributed over three software modules.

Firedrake This is the framework for the automated solution of finite element methods described in Section **??**.

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

by PyOP2 (see Section 2.2.2).

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

The interaction among these modules is illustrated in Figure **??**.

There are several reasons behind the choice of this three-layer structuring:

Simpler analysis The abstractions used in Firedrake and PyOP2 simplify the analysis required for automation. For example, from the parallel loop construct in PyOP2 we can derive sufficient information to build a loop chain (see Section 3.5.3).

User base We wish generalized sparse tiling to be used in real scientific software, so we targeted a system with a user base in steady expansion. This increased the implementation effort, but it has also made sparse tiling promptly available to a wide audience.

Flexibility Adopting sparse tiling in a Firedrake program is the simplest option, because the generation of inspector/executor schemes is fully automated. If we instead have a PyOP2 program, the generation of inspector/executor schemes is only partly automated, since the separation of a set into the core, boundary and non-exec regions for distributed-memory parallelism must be coded explicitly. Users can also write an inspector-executor scheme from scratch, in which case the entire loop chain must be provided through direct calls to the SLOPE library (an example was provided in Listing 7). This is clearly the most flexible yet complicated option.

In the next sections, we describe the interplay between these three frameworks and how this leads to automation.

3.8.1 SLOPE: a Library for Tiling Irregular Computations

SLOPE is an open source software that provides a set of functions to build a loop chain and to write inspector/executor schemes for sparse tiling[3].

[3]SLOPE is freely available at `https://github.com/coneoproject/SLOPE`

The loop chain abstraction implemented by SLOPE is the one described in Section 3.5.3. In essence, a loop chain comprises some sets (including the separation into core, boundary, and non-exec regions), maps between sets, and a sequence of loops. Each loop has one or more descriptors specifying what and how dataset are accessed. The example in Listing **??** illustrates the interface exposed by SLOPE for constructing loop chains.

SLOPE implements all of the algorithms in Section 3.7.2. It also provides features to verify the effectiveness and the correctness of the transformations:

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

The summary routine The inspector returns useful information concerning the tiling process, including: the number and the average size of tiles, the total number of colors used (which can give an indication of how effective a shared-memory parallelization will be), times for the most critical 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.

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

PyOP2 has been described in Section 2.2.2. This section focuses on three complementary aspects: (i) the interface with the user for identifying sequences of "chainable" loops; (ii) the lazy evaluation mechanism that allows building loop chains; (iii) the interface with SLOPE to run an inspector/executor scheme.

To trigger the creation of a loop chain, PyOP2 provides the *loop_chain* construct. The use of this interface is exemplified in Listing **??**. The

loop_chain interface is exposed to both users and higher layers of abstraction. Thus, in Firedrake we define a loop chain in a way analogous to PyOP2's, with the only difference being that loops are not visible in the source code because they are automatically generated from the mathematical notation.

The second fundamental aspect is that PyOP2 implements lazy evaluation of parallel loops. The parallel loops encountered or generated by Firedrake are first pushed into a queue. 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 recent parallel loop to the oldest one – and a new sub-trace produced. The sub-trace, which includes all parallel loops that need be executed for a correct update of f, is finally executed.

All loops in the trace that are created within a *loop_chain* scope are sparse tiled through SLOPE. In detail, the interaction between PyOP2 and SLOPE is 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 executes 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 consists of two main steps: "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 all relevant data structures (sets, maps, loops) into a format suitable for SLOPE. Eventually, a string of C code implementing an inspector for the loops in the *loop_chain* is returned. PyOP2 just-in-

time compiles this code and runs it, obtaining the *inspection* data structure.

4. A "software cache" mapping *loop_chain*s to *inspection* data structures is used. This whole process is therefore executed only once for each unique *loop_chain* encountered in the code.

5. The executor is built analogously, and invoked each time a *loop_chain* is exited.

3.8.3 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, is completely handled in 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 2.2.2, PyOP2 sets distinguish between different regions: core, owned, exec, and non-exec. Firedrake is directly responsible for setting these regions and creating data structures enabling message passing among different processes.

To support our loop chain abstraction, Firedrake must be able to allocate arbitrarily deep halo regions; both Firedrake and DMPlex have been extended with this feature[4]. A parameter called *S-depth* (for historical reasons, see **?**), used for initializing a mesh, specifies the extent of the halo regions. A value of *S-depth* equal to n indicates the presence of n strips of off-process data elements in each set. The default value for *S-depth* is 1, which corresponds to the standard execution model of Firedrake/PyOP2. Even in absence of sparse tiling, this value enables overlap of computation with communication, at the price of a small amount of redundant computation along partition boundaries.

[4]Michael Lange carried out the bulk of the implementation.

3.9 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[5]. The aim of this phase was to evaluate the performance improvements in a real-world simulation.

3.9.1 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 Strout et al. [2014], 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.

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[6]. Given a sparse matrix A, and a vector $\vec{f}$, related by $A\vec{u} = \vec{f}$,

[5]Seigen has been developed by Christian Jacobs.

[6]This section is partly extracted from Strout et al. [2014], and the results were obtained from experiments conducted by Christopher D. Krieger, Catherine Olschanowsky, and Michelle Mills Strout

Matrix name	Execution time reduction	Speed-up
ldoor	40.34	12.11
pwtk	38.42	11.98
thermal2	25.78	11.08
xenon2	20.15	9.53
audikw_1	13.42	8.70
nd24k	-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.

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 Davis and Hu [2011], was used. The matrices were selected so that they would vary in overall data footprint, from 45 MB to 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

Implementation	Execution time	Speed-up
Westmere *omp*	36.87	6.43
Westmere *mpi*	31.0	7.66
Westmere *tiled*	26.49	8.96
Sandy Bridge *omp*	30.01	6.65
Sandy Bridge *mpi*	24.42	8.17
Sandy Bridge *tiled*	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).

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 3.9.1.

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 Giles et al. [2011]. The *tiled* implementation uses an early version of the SLOPE library (the differences with the inspector algorithms shown in Section 3.7.2 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 *tiled* versions suffer from the well-known NUMA effect as threads are always equally spread across the two sockets. Nevertheless, compared to *mpi*, the *tiled* version exhibits a peak runtime reduction of 15% on the Westmere and of 16% on the Sandy Bridge.

Results shown for *tiled* 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 overall 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 3.7.2. 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 3.8 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 3.5.3) and inspector algorithms (Section 3.7.2), for supporting MPI-based executors.

3.9.2 Seigen: an Elastic Wave Equation Solver for Seismological Problems

...

3.10 Conclusions and Future Work

Chapter 4

Minimizing the Operation Count of Finite Element Integration Loops

In this chapter, we present an algorithm for the optimization of a class of finite element integration loop nests. This algorithm, which exploits fundamental mathematical properties of finite element operators, is proven to achieve a locally optimal operation count. In specified circumstances the optimum achieved is global. Extensive numerical experiments demonstrate significant performance improvements over the state of the art in finite element code generation in almost all cases. This validates the effectiveness of the algorithm presented here, and illustrates its limitations. The algorithm, as explained in later sections, is implemented in Firedrake, the framework for the automated resolution of PDEs via the finite element method reviewed in Section 2.2.1.

4.1 Motivation and Related Work

The need for rapid implementation of high performance, robust, and portable finite element methods has led to approaches based on automated code generation. This has been proven successful in the context of the FEniCS and Firedrake projects. In these frameworks, the weak variational form of a problem is expressed in a high level mathematical syntax by means of the domain-specific language UFL **?**. This mathematical spec-

ification is used by a domain-specific compiler, known as a form compiler, to generate low-level C or C++ code for the integration over a single element of the computational mesh of the variational problem's left and right hand side operators. The code for assembly operators must be carefully optimized: as the complexity of a variational form increases, in terms of number of derivatives, pre-multiplying functions, or polynomial order of the chosen function spaces, the operation count increases, with the result that assembly often accounts for a significant fraction of the overall runtime. This aspect has previously been introduced in Section 2.1.5.

As demonstrated by the substantial body of research on the topic, automating the generation of such high performance implementations poses several challenges. This is a result of the complexity inherent in the mathematical expressions involved in the numerical integration, which varies from problem to problem, and the particular structure of the loop nests enclosing the integrals. General-purpose compilers, such as those by *GNU* and *Intel*, fail to exploit the structure inherent in the expressions, thus producing sub-optimal code (i.e., code which performs more floating-point operations, or "flops", than necessary; we show this in Section 4.6). Research compilers, for instance those based on polyhedral analysis of loop nests, such as PLUTO **?**, focus on parallelization and optimization for cache locality, treating issues orthogonal to the question of minimising flops. The lack of suitable third-party tools has led to the development of a number of domain-specific code transformation (or synthesizer) systems. **?** show how automated code generation can be leveraged to introduce optimizations that a user should not be expected to write "by hand". **?** and **?** employ mathematical reformulations of finite element integration with the aim of minimizing the operation count. In **?** – as well as in Chapter 5, from which the article has been derived – the effects and the interplay of generalized code motion and a set of low level optimizations are analysed. It is also worth mentioning two new new form compilers, UFLACS **?** and TSFC **?**, which particularly target the compilation time challenges of the more complex variational forms. The performance evaluation in Section 4.6 includes most of these systems.

However, in spite of such a considerable research effort, there is still no answer to one fundamental question: can we automatically generate an implementation of a form which is optimal in the number of flops exe-

cuted? In this chapter, we formulate an approach that solves this problem for a particular class of forms and provides very good approximations in all other cases. In particular, we will define "local optimality", which relates operation count with inner loops. In summary, our contributions are as follows:

- We formalize the class of finite element integration loop nests and we build the space of legal transformations impacting their operation count.
- We provide an algorithm to select points in the transformation space. The algorithm uses a cost model to: (i) understand whether a transformation reduces or increases the operation count; (ii) choose between different (non-composable) transformations.
- We demonstrate that our approach systematically leads to a local optimum. We also explain under what conditions of the input problem global optimality is achieved.
- We integrate our approach with a compiler, COFFEE[1], which is in use in the Firedrake framework. The structure of COFFEE is discussed in Chapter 6.
- We experimentally evaluate using a broader suite of forms, discretizations, and code generation systems than has been used in prior research. This is essential to demonstrate that our optimality model holds in practice.

In addition, in order to place COFFEE on the same level as other code generation systems from the viewpoint of low level optimization, which is essential for a fair performance comparison:

- We introduce a transformation based on symbolic execution that allows irrelevant floating point operations to be skipped (for example those involving zero-valued quantities).

In Section 4.2.1 we introduce a set of definitions mapping mathematical properties to the level of loop nests. This step is an essential precursor to

[1]COFFEE stands for COmpiler For Fast Expression Evaluation. The compiler is open-source and available at https://github.com/coneoproject/COFFEE

the definition of the two algorithms – sharing elimination (Section 4.2.2) and pre-evaluation (Section 4.2.3) – through which we construct the space of legal transformations. The main transformation algorithm in Section 4.3 delivers the local optimality claim by using a cost model to coordinate the application of sharing elimination and pre-evaluation. We elaborate on the correctness of the methodology in Section 4.4. The numerical experiments are showed in Section 4.6. We conclude discussing the limitations of the algorithms presented and future work.

4.2 Transformation Space

In this section, we characterize global and local optimality for finite element integration as well as the space of legal transformations that needs be explored to achieve them. The method by which exploration is performed is discussed in Section 4.3.

4.2.1 Loop nests, expressions and optimality

In order to make the article self-contained, we start with reviewing basic compiler terminology.

Definition 1 (Perfect and imperfect loop nests)**.** *A perfect loop nest is a loop whose body either 1) comprises only a sequence of non-loop statements or 2) is itself a perfect loop nest. If this condition does not hold, a loop nest is said to be imperfect.*

Definition 2 (Independent basic block)**.** *An independent basic block is a sequence of statements such that no data dependencies exist between statements in the block.*

We focus on perfect nests whose innermost loop body is an independent basic block. A straightforward property of this class is that hoisting invariant expressions from the innermost to any of the outer loops or the preheader (i.e., the block that precedes the entry point of the nest) is always safe, as long as any dependencies on loop indices are honored. We will make use of this property. The results of this section could also be generalized to larger classes of loop nests, in which basic block indepen-

dence does not hold, although this would require refinements beyond the scope of this paper.

By mapping mathematical properties to the loop nest level, we introduce the concepts of a *linear loop* and, more generally, a (perfect) multilinear loop nest.

Definition 3 (Linear loop)**.** *A loop L defining the iteration space I through the iteration variable i, or simply L_i, is linear if in its body*

1. *i appears only as an array index, and*
2. *whenever an array a is indexed by i ($a[i]$), all expressions in which this appears are affine in $a[i]$.*

Definition 4 (Multilinear loop nest)**.** *A multilinear loop nest of arity n is a perfect nest composed of n loops, in which all of the expressions appearing in the body of the innermost loop are affine in each loop L_i separately.*

We will show that multilinear loop nests, which arise naturally when translating bilinear or linear forms into code, are important because they have a structure that we can take advantage of to reach a local optimum.

We define two other classes of loops.

Definition 5 (Reduction loop)**.** *A loop L_i is said to be a reduction loop if in its body*

1. *i appears only as an array index, and*
2. *for each augmented assignment statement S (e.g., an increment), arrays indexed by i appear only on the right hand side of S.*

Definition 6 (Order-free loop)**.** *A loop L_i is said to be an order-free loop if its iterations can be executed in any arbitrary order.*

Consider Equation 2.21 and the (abstract) loop nest implementing it illustrated in Figure 4.1. The imperfect nest $\Lambda = [L_e, L_i, L_j, L_k]$ comprises an order-free loop L_e (over elements in the mesh), a reduction loop L_i (performing numerical integration), and a multilinear loop nest $[L_j, L_k]$ (over test and trial functions). In the body of L_k, one or more statements evaluate the local tensor for the element e. Expressions (the right hand side of a statement) result from the translation of a form in high level matrix

```
for (e = 0; e < E; e++)
  ...
  for (i = 0; i < I; i++)
    ...
    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
```
$$a_{ejk} \mathrel{+}= \sum_{w=1}^{m} \alpha_{eij}^{w} \beta_{eik}^{w} \sigma_{ei}^{w}$$

Figure 4.1: The loop nest implementing a generic bilinear form.

notation into code. In particular, m is the number of monomials (a form is a sum of monomials), α_{eij} (β_{eik}) represents the product of a coefficient function (e.g., the inverse Jacobian matrix for the change of coordinates) with test or trial functions, and σ_{ei} is a function of coefficients and geometry. We do not pose any restrictions on function spaces (e.g., scalar- or vector-valued), coefficient expressions (linear or non-linear), differential and vector operators, so σ_{ei} can be arbitrarily complex. We say that such an expression is in *normal form*, because the algebraic structure of a variational form is intact: products have not yet been expanded, distinct monomials can still be identified, and so on. This brings us to formalize the class of loop nests that we aim to optimize.

Definition 7 (Finite element integration loop nest)**.** *A finite element integration loop nest is a loop nest in which the following appear, in order: an imperfect order-free loop, an imperfect (perfect only in some special cases), linear or non-linear reduction loop, and a multilinear loop nest whose body is an independent basic block in which expressions are in normal form.*

We then characterize optimality for a finite element integration loop nest as follows.

Definition 8 (Optimality of a loop nest)**.** *Let Λ be a generic loop nest, and let Γ be a transformation function $\Gamma : \Lambda \to \Lambda'$ such that Λ' is semantically equivalent to Λ (possibly, $\Lambda' = \Lambda$). We say that $\Lambda' = \Gamma(\Lambda)$ is an optimal synthesis of Λ if the total number of operations (additions, products) performed to evaluate the result is minimal.*

The concept of local optimality, which relies on the particular class of *flop-decreasing* transformations, is also introduced.

Definition 9 (Flop-decreasing transformation)**.** *A transformation which reduces the operation count is called flop-decreasing.*

Definition 10 (Local optimality of a loop nest)**.** *Given* Λ*,* Λ' *and* Γ *as in Definition 8, we say that* $\Lambda' = \Gamma(\Lambda)$ *is a locally optimal synthesis of* Λ *if:*

- *the number of operations (additions, products) in the innermost loops performed to evaluate the result is minimal, and*
- Γ *is expressed as composition of flop-decreasing transformations.*

The restriction to flop-decreasing transformations aims to exclude those apparent optimizations that, to achieve flop-optimal innermost loops, would rearrange the computation at the level of the outer loops causing, in fact, a global increase in operation count.

We also observe that Definitions 8 and 10 do not take into account memory requirements. If the execution of loop nest were memory-bound – the ratio of operations to bytes transferred from memory to the CPU being too low – then optimizing the number of flops would be fruitless. Henceforth we assume we operate in a CPU-bound regime, evaluating arithmetic-intensive expressions. In the context of finite elements, this is often true for more complex multilinear forms and/or higher order elements.

Achieving optimality in polynomial time is not generally feasible, since the σ_{ei} sub-expressions can be arbitrarily unstructured. However, multilinearity results in a certain degree of regularity in α_{eij} and β_{eik}. In the following sections, we will elaborate on these observations and formulate an approach that achieves: (i) at least a local optimum in all cases; (ii) global optimality whenever the monomials are "sufficiently structured". To this purpose, we will construct:

- the space of legal transformations impacting the operation count (Sections 4.2.2 – 4.2.4)
- an algorithm to select points in the transformation space (Section 4.3)

4.2.2 Sharing elimination

We start with introducing the fundamental notion of sharing.

Definition 11 (Sharing)**.** *A statement within a loop nest* Λ *presents sharing if at least one of the following conditions hold:*

Spatial sharing *There are at least two symbolically identical sub-expressions*

Temporal sharing *There is at least one non-trivial sub-expression (e.g., an addition or a product) that is redundantly executed because it is independent of* $\{L_{i_1}, L_{i_1}, ...L_{i_n}\} \subset \Lambda$.

To illustrate the definition, we show in Figure 4.2 how sharing evolves as factorization and code motion are applied to a trivial multilinear loop nest. In the original loop nest (Figure 4.2a), spatial sharing is induced by the symbol b_j. Factorization eliminates spatial sharing and creates temporal sharing (Figure 4.2b). Finally, generalized code motion **?**, which hoists sub-expressions that are redundantly executed by at least one loop in the nest[2], leads to optimality (Figure 4.2c).

```
for (j = 0; j < J; j++)
  for (i = 0; i < I; i++)
    a_ji += b_j c_i + b_j d_i
```

(a) With spatial sharing

```
for (j = 0; j < J; j++)
  for (i = 0; i < I; i++)
    a_ji += b_j (c_i + d_i)
```

(b) With temporal sharing

```
for (i = 0; i < I; i++)
  t_i = c_i + d_i
for (j = 0; j < J; j++)
  for (i = 0; i < I; i++)
    a_ji += b_j t_i
```

(c) Optimal form

Figure 4.2: Reducing a simple multilinear loop nest to optimal form.

In this section, we study *sharing elimination*, a transformation that aims to reduce the operation count by removing sharing through the application of expansion, factorization, and generalized code motion. If the objective were reaching optimality and the expressions lacked structure, a transformation of this sort would require solving a large combinatorial problem – for instance to evaluate the impact of all possible factorizations. Our sharing elimination strategy, instead, exploits the structure inherent in finite element integration expressions to guarantee, after coordination with other transformations (an aspect which we discuss in the following sections), local optimality. Global optimality is achieved if stronger preconditions hold. Setting local optimality, rather than optimality, as primary goal is essential to produce simple and computationally efficient algorithms – two necessary conditions for integration with a compiler.

[2]Traditional loop-invariant code motion, which is commonly applied by general-purpose compilers, only checks invariance with respect to the innermost loop.

Identification and exploitation of structure

Finite element expressions can be seen as composition of operations between tensors. Often, the optimal implementation strategy for these operations is to be determined out of two alternatives. For instance, consider $J^{-T}\nabla v \cdot J^{-T}\nabla v$, with J^{-T} being the transposed inverse Jacobian matrix for the change of (two-dimensional) coordinates, and v a generic two-dimensional vector. The tensor operation will reduce to the scalar expression $(av_i^0 + bv_i^1)(av_i^0 + bv_i^1) + ...$, in which v_i^0 and v_i^1 represent components of v that depend on L_i. To minimize the operation count for expressions of this kind, we have two options:

Strategy 1. *Eliminating temporal sharing through generalized code motion.*

Strategy 2. *Eliminating spatial sharing first – through product expansion and factorization – and temporal sharing afterwards, again through generalized code motion.*

In the current example, we observe that, depending on the size of L_i, applying Strategy 2 could reduce the operation count since the expression would be recast as $v_i^0 v_i^0 aa + v_i^0 v_i^1 (ab + ab) + v_i^1 v_i^1 cc + ...$ and some hoistable sub-expressions would be exposed. On the other hand, Strategy 1 would have no effect as v only depends on a single loop, L_i. In general, the choice between the two strategies depends on multiple factors: the loop sizes, the increase in operation count due to expansion (in Strategy 2), and the gain due to code motion. A second application of Strategy 2 was provided in Figure 4.2. These examples motivate the introduction of a particular class of expressions, for which the two strategies assume notable importance.

Definition 12 (Structured expression)**.** *We say that an expression is "structured along a loop nest Λ" if and only if, for every symbol s_Λ depending on at least one loop in Λ, the spatial sharing of s_Λ may be eliminated by factorizing all occurrences of s_Λ in the expression.*

Proposition 1. *An expression along a multilinear loop nest is structured.*

Proof. This follows directly from Definition 3 and Definition 4, which essentially restrict the number of occurrences of a symbol s_Λ in a summand to at most 1. □

If Λ were an arbitrary loop nest, a given symbol s_Λ could appear everywhere (e.g., n times in a summand and m times in another summand with $n \neq m$, as argument of a higher level function, in the denominator of a division), thus posing the challenge of finding the factorization that maximizes temporal sharing. If Λ is instead a finite element integration loop nest, thanks to Proposition 1 the space of flop-decreasing transformations is constructed by "composition" of Strategy 1 and Strategy 2, as illustrated in Algorithm 1.

Finally, we observe that the σ_{ei} sub-expressions can sometimes be considered "weakly structured". This happens when a relaxed version of Definition 12 applies, in which the factorization of s_Λ only "minimizes" (rather than "eliminates") spatial sharing (for instance, in the complex hyperelastic model analyzed in Section 4.6). Weak structure will be exploited by Algorithm 1 in the attempt to achieve optimality.

Algorithm

Algorithm 1 describes sharing elimination assuming as input a tree representation of the loop nest. It makes use of the following notation and terminology:

- *multilinear operand*: any α_{eij} or β_{eik} in the input expression.
- *multilinear symbol*: a symbol appearing within a multilinear operand depending on L_j or L_k (e.g., test functions, first order derivatives of test functions, etc.).

Examples will be provided in Section 4.2.2.

Algorithm 1 (Sharing elimination)**.** The input of the algorithm is a tree representation a finite element integration loop nest.

1. Perform a depth-first visit of the loop tree to collect and partition multilinear operands into disjoint sets, $\mathbb{P} = \{P_1, ..., P_p\}$. $\mathbb{P}$ is such that all multilinear operands in each $P \in \mathbb{P}$ share the same set of multilinear symbols S_P, whereas there is no sharing across different partitions. For all multilinear operands in $P \in \mathbb{P}$ such that $|P| \leq |S_P|$, apply Strategy 1.

Note: as a consequence of Proposition 1, $|P|$ and $|S_P|$ represent the number of products in the innermost loop induced by P if Strategy 1 or Strategy 2 were applied

2. For each sub-expression expr depending on exactly one linear loop, collect the multilinear symbols and the temporaries produced at step (1). Partition them into disjoint sets, $\mathbb{T} = \{T_1, ..., T_t\}$, such that T_i includes all instances of a given symbol in expr. Apply Strategy 2 factorizing the symbols in each T_i, provided that this leads to a reduction in operation count; otherwise, apply Strategy 1
   Note: the last check ensures the flop-decreasing nature of the transformation. In the cases in which expansion outweighs code motion, Strategy 1 is preferred.
   Note: the expansion cost is a function of the products wrapping a symbol (how many of them and their arity), so it can be determined through tree visits.

3. Build the *sharing graph* $G = (S, E)$. Each $s \in S$ represents a multilinear symbol or a temporary produced by the previous steps. An edge (s_i, s_j) indicates that a product $s_i s_j$ would appear if the sub-expressions including s_i and s_j were expanded.
   Note: the following steps will only impact bilinear forms, since otherwise $E = \emptyset$.

4. Partition S into disjoint sets, $\mathbb{S} = \{S_1, ..., S_n\}$, such that S_i includes all instances of a given symbol s in the expression. Transform G by merging $\{s_1, ..., s_m\} \subset S_i$ into a unique vertex s (taking the union of the edges), provided that factorizing $[s_1, ..., s_m]$ would not cause an increase in operation count.

5. Map G to an Integer Linear Programming (ILP) model for determining how to optimally apply Strategy 2. The solution is the set of symbols that will be factorized by Strategy 2. Let $|S| = n$; the ILP

model then is as follows:

x_i: a vertex in S (1 if a symbol should be factorized, 0 otherwise)
y_{ij}: an edge in E (1 if s_i is factorized in the product $s_i s_j$, 0 otherwise)
n_i: the number of edges incident to x_i

$$\min \sum_{i=1}^{n} x_i, \ s.t. \qquad \sum_{j|(i,j)\in E} y_{ij} \leq n_i x_i, \ i = 1, ..., n$$
$$y_{ij} + y_{ji} = 1, \ (i,j) \in E$$

6. Perform a depth-first visit of the loop tree and, for each yet unhandled or hoisted expression, apply the most profitable between Strategy 1 and Strategy 2.
   Note: this pass speculatively assumes that expressions are (weakly) structured along the reduction loop. If the assumption does not hold, the operation count will generally be sub-optimal because only a subset of factorizations and code motion opportunities may eventually be considered.

Although the primary goal of Algorithm 1 is operation count minimization within the multilinear loop nest, the enforcement of flop-decreasing transformations (steps (2) and (4)) and the re-scheduling of sub-expressions within outer loops (last step) also attempt to optimize the loop nest globally. We will further elaborate this aspect in Section 4.4.

Examples

Consider again Figure 4.2a. We have $\mathbb{P} = \{P_0, P_1, P_2\}$, with $P_0 = \{b_j\}$, $P_1 = \{c_i\}$, and $P_2 = \{d_i\}$. For all P_i, we have $|P_i| = 1 = |S_{P_i}|$, although applying Strategy 1 in step (1) has no effect. The sharing graph is $G = (\{b_j, c_i, d_i\}, \{(b_j, c_i), (b_j, d_i)\}$, and $T = \{c_i, d_i\}$. The ILP formulation leads to the code in Figure 4.2c.

In Figure 4.3, Algorithm 1 is executed in a very simple realistic scenario, which originates from the bilinear form of a Poisson equation in two dimensions. We observe that $\mathbb{P} = \{P_0, P_1\}$, with $P_0 = \{(z_0 a_{ik} + z_2 b_{ik}), (z_1 a_{ik} + z_3 b_{ik})\}$ and $P_1 = \{(z_0 a_{ij} + z_2 b_{ij}), (z_1 a_{ij} + z_3 b_{ij})\}$. In addition, $|P_i| = |S_{P_i}| = 2$, so Strategy 1 is applied to both partitions (step (1)).

We then have (step (3)) $G = (\{t_0, t_1, t_2, t_3\}, \{(t_0, t_2), (t_1, t_3)\})$. Since there are no more factorization opportunities, the ILP formulation becomes irrelevant.

```
for (e = 0; e < E; e++)
  z0 = ...
  z1 = ...
  ...
  for (i = 0; i < I; i++)
    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        a_ejk += (((z0*a_ik + z2*b_ik)*
                   (z0*c_ij + z2*d_ij))+
                  ((z1*a_ik + z3*b_ik)*
                   (z1*c_ij + z3*d_ij)))*
                 W_i * det
```

(a) Normal form

```
for (e = 0; e < E; e++)
  ...
  for (i = 0; i < I; i++)
    for (k = 0; k < K; k++)
      t0_k = (z0*a_ik + z2*b_ik)
      t1_k = (z1*a_ik + z3*b_ik)
    for (j = 0; j < J; j++)
      t2_j = (z0*c_ij + z2*d_ij) * W_i * det
      t3_j = (z1*c_ij + z3*d_ij) * W_i * det
    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        a_ejk += t0_k * t2_j + t1_k * t3_j
```

(b) After sharing elimination

Figure 4.3: Applying sharing elimination to the bilinear form arising from a Poisson equation in 2D. The operation counts are $E(f(z_0, z_1, ...) + IJK \cdot 18)$ (left) and $E(f(z_0, z_1, ...) + I(J \cdot 6 + K \cdot 9 + JK \cdot 4)$ (right), with $f(z_0, z_1, ...)$ representing the operation count for evaluating $z_0, z_1, ...$, and common sub-expressions being counted once. The synthesis in Figure 4.3b is globally optimal apart from the pathological case $I, J, K = 1$.

4.2.3 Pre-evaluation of reductions

Sharing elimination uses three operators: expansion, factorization, and code motion. In this section, we discuss the role and legality of a fourth operator: reduction pre-evaluation. We will see that what makes this operator special is the fact that there exists a single point in the transformation space of a monomial (i.e., a specific factorization of test, trial, and coefficient functions) ensuring its correctness.

We start with an example. Consider again the loop nest and the expression in Figure 4.1. We pose the following question: are we able to identify sub-expressions for which the reduction induced by L_i can be pre-evaluated, thus obtaining a decrease in operation count proportional to the size of L_i, I? The transformation we look for is exemplified in Figure 4.4 with a simple loop nest. The reader may verify that a similar transformation is applicable to the example in Figure 4.3a.

```
for (e = 0; e < E; e++)
  for (i = 0; i < I; i++)
    for (k = 0; k < K; k++)
      a_ek += d_e b_ik c_i + d_e b_ik d_i
```

(a) With reduction

```
for (i = 0; i < I; i++)
  for (k = 0; k < K; k++)
    t_k += b_ik (c_i + d_i)

for (e = 0; e < E; e++)
  for (k = 0; k < K; k++)
    a_ek = d_e t_k
```

(b) After pre-evaluation

Figure 4.4: Exposing (through factorization) and pre-evaluating a reduction.

Pre-evaluation can be seen as the generalization of tensor contraction (Section **??**) to a wider class of sub-expressions. We know that multilinear forms can be seen as sums of monomials, each monomial being an integral over the equation domain of products (of derivatives) of functions from discrete spaces. A monomial can always be reduced to the product between a "reference" and a "geometry" tensor. In our model, a reference tensor is simply represented by one or more sub-expressions independent of L_e, exposed after particular transformations of the expression tree. This leads to the following algorithm.

Algorithm 2 (Pre-evaluation)**.** Consider a finite element integration loop nest $\Lambda = [L_e, L_i, L_j, L_k]$. We dissect the normal form input expression into distinct sub-expressions, each of them representing a monomial. Each sub-expression is then factorized so as to split constants from $[L_i, L_j, L_k]$-dependent terms. This transformation is feasible[3], as a consequence of the results in **?**. These $[L_i, L_j, L_k]$-dependent terms are hoisted outside of Λ and stored into temporaries. As part of this process, the reduction induced by L_i is computed by means of symbolic execution. Finally, L_i is removed from Λ.

The pre-evaluation of a monomial introduces some critical issues:

1. Depending on the complexity of a monomial, a certain number, t, of temporary variables is required if pre-evaluation is performed. Such

[3]For reasons of space, we omit the detailed sequence of steps (e.g., expansion, factorization), which is however available at https://github.com/coneoproject/COFFEE/blob/master/coffee/optimizer.py in **?**.

temporary variables are actually n-dimensional arrays of size S, with n and S being, respectively, the arity and the extent (iteration space size) of the multilinear loop nest (e.g., $n = 2$ and $S = JK$ in the case of bilinear forms). For certain values of $<t, n, S>$, pre-evaluation may dramatically increase the working set, which may be counter-productive for actual execution time.

2. The transformations exposing $[L_i, L_j, L_k]$-dependent terms increase the arithmetic complexity of the expression (e.g., expansion tends to increase the operation count). This could outweigh the gain due to pre-evaluation.

3. A strategy for coordinating sharing elimination and pre-evaluation is needed. We observe that sharing elimination inhibits pre-evaluation, whereas pre-evaluation could expose further sharing elimination opportunities.

We expand on point (1) in the next section, while we address points (2) and (3) in Section 4.3.

4.2.4 Memory constraints

We have just observed that the code motion induced by monomial pre-evaluation may dramatically increase the working set size. Even more aggressive code motion strategies are theoretically conceivable. Imagine Λ is enclosed in a time stepping loop. One could think of exposing (through some transformations) and hoisting time-invariant sub-expressions for minimizing redundant computation at each time step. The working set size would then increase by a factor E, and since $E \gg I, J, K$, the gain in operation count would probably be outweighed, from a runtime viewpoint, by a much larger memory pressure.

Since, for certain forms and discretizations, hoisting may cause the working set to exceed the size of some level of local memory (e.g. the last level of private cache on a conventional CPU, the shared memory on a GPU), we introduce the following *memory constraints*.

Constraint 1. *The size of a temporary due to code motion must not be proportional to the size of L_e.*

Constraint 2. *The total amount of memory occupied by the temporaries due to code motion must not exceed a certain threshold, T_H.*

Constraint 1 is a policy decision that the compiler should not silently consume memory on global data objects. It has the effect of shrinking the transformation space. Constraint 2 has both theoretical and practical implications, which will be carefully analyzed in the next sections.

4.3 Selection and composition of transformations

In this section, we build a transformation algorithm that, given a memory bound, systematically reaches a local optimum for finite element integration loop nests.

4.3.1 Transformation algorithm

We address the two following issues:

1. *Coordination of pre-evaluation and sharing elimination.* Recall from Section 4.2.3 that pre-evaluation could either increase or decrease the operation count in comparison with that achieved by sharing elimination.

2. *Optimizing over composite operations.* Consider a form comprising two monomials m_1 and m_2. Assume that pre-evaluation is profitable for m_1 but not for m_2, and that m_1 and m_2 share at least one term (for example some basis functions). If pre-evaluation were applied to m_1, sharing between m_1 and m_2 would be lost. We then need a mechanism to understand which transformation – pre-evaluation or sharing elimination – results in the highest operation count reduction when considering the whole set of monomials (i.e., the expression as a whole).

Let $\theta : M \to \mathbb{Z}$ be a cost function that, given a monomial $m \in M$, returns the gain/loss achieved by pre-evaluation over sharing elimination. In particular, we define $\theta(m) = \grave{}^{pre}(m) - \grave{}^{se}(m)$, where $\grave{}^{se}$ and $\grave{}^{pre}$ represent the operation counts resulting from applying sharing elimination and pre-evaluation, respectively. Thus pre-evaluation is profitable for m

if and only if $\theta(m) < 0$. We return to the issue of deriving $\grave{}^{se}$ and $\grave{}^{pre}$ in Section 4.3.2. Having defined θ, we can now describe the transformation algorithm (Algorithm 3).

Algorithm 3 (Transformation algorithm)**.** The algorithm has three main phases: initialization (step 1); determination of the monomials preserving the memory constraints that should be pre-evaluated (steps 2-4); application of pre-evaluation and sharing elimination (step 5).

1. Perform a depth-first visit of the expression tree and determine the set of monomials M. Let S be the subset of monomials m such that $\theta(m) > 0$. The set of monomials that will *potentially* be pre-evaluated is $P = M \setminus S$.
   Note: there are two fundamental reasons for not pre-evaluating $m_1 \in P$ straight away: 1) the potential presence of spatial sharing between m_1 and $m_2 \in S$, which impacts the search for the global optimum; 2) the risk of breaking Constraint 2.

2. Build the set B of all possible bipartitions of P. Let D be the dictionary that will store the operation counts of different alternatives.

3. Discard $b = (b_S, b_P) \in B$ if the memory required after applying pre-evaluation to the monomials in b_P exceeds T_H (see Constraint 2); otherwise, add $D[b] = \grave{}^{se}(S \cup b_S) + \grave{}^{pre}(b_P)$.
   Note: $\mathbb{B}$ is in practice very small, since even complex forms usually have only a few monomials. This pass can then be accomplished rapidly as long as the cost of calculating $\grave{}^{se}$ and $\grave{}^{pre}$ is negligible. We elaborate on this aspect in Section 4.3.2.

4. Take $\arg\min_b D[b]$.

5. Apply pre-evaluation to all monomials in b_P. Apply sharing elimination to all resulting expressions.
   Note: because of the reuse of basis functions, pre-evaluation may produce some identical tables, which will be mapped to the same temporary variable. Sharing elimination is therefore transparently applied to all expressions, including those resulting from pre-evaluation.

The output of the transformation algorithm is provided in Figure 4.5, assuming as input the loop nest in Figure 4.1.

```
// Pre-evaluated tables
...
for (e = 0; e < E; e++)
  // Temporaries due to sharing elimination
  // (Sharing was a by-product of pre-evaluation)
  ...
  // Loop nest for pre-evaluated monomials
  for (j = 0; j < J; j++)
    for (k = 0; k < K; k++)
      a_ejk += F'(...) + F''(...) + ...

  // Loop nest for monomials for which run-time
  // integration was determined to be faster
  for (i = 0; i < I; i++)
    // Temporaries due to sharing elimination
    ...
    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        a_ejk += H(...)
```

Figure 4.5: The loop nest produced by the algorithm for an input as in Figure 4.1.

4.3.2 The cost function θ

We tie up the remaining loose end: the construction of the cost function θ.

We recall that $\theta(m) = \grave{}^{se}(m) - \grave{}^{pre}(m)$, with $\grave{}^{se}$ and $\grave{}^{pre}$ representing the operation counts after applying sharing elimination and pre-evaluation. Since θ is deployed in a working compiler, simplicity and efficiency are essential characteristics. In the following, we explain how to derive these two values.

The most trivial way of evaluating $\grave{}^{se}$ and $\grave{}^{pre}$ would consist of applying the actual transformations and simply count the number of operations. This would be tolerable for $\grave{}^{se}$, as Algorithm 1 tends to have negligible cost. However, the overhead would be unacceptable if we applied pre-evaluation – in particular, symbolic execution – to all bipartitions analyzed by Algorithm 3. We therefore seek an analytic way of determining $\grave{}^{pre}$.

The first step consists of estimating the *increase factor*, ι. This number captures the increase in arithmetic complexity due to the transfor-

mations exposing pre-evaluation opportunities. For context, consider the example in Figure 4.6. One can think of this as the (simplified) loop nest originating from the integration of the action of a mass matrix. The sub-expression f_0*B_{i0}+f_1*B_{i1}+f_2*B_{i2} represents the coefficient f over (tabulated) basis functions (array B). In order to apply pre-evaluation, the expression needs be transformed to separate f from all $[L_i, L_j, L_k]$-dependent quantities (see Algorithm 2). By product expansion, we observe an increase in the number of $[L_j, L_k]$-dependent terms of a factor $\iota = 3$.

```
for (i = 0; i < I; i++)
  for (j = 0; j < J; j++)
    for (k = 0; k < K; k++)
      a_jk += b_ij*b_ik*(f_0*B_i0+f_1*B_i1+f_2*B_i2)
```

Figure 4.6: Simplified loop nest for a pre-multiplied mass matrix.

In general, however, determining ι is not so straightforward since redundant tabulations may result from common sub-expressions. Consider the previous example. One may add one coefficient in the same function space as f, repeat the expansion, and observe that multiple sub-expressions (e.g., $b_{10} * b_{01} * ...$ and $b_{01} * b_{10} * ...$) will reduce to identical tables. To evaluate ι, we then use combinatorics. We calculate the k-combinations with repetitions of n elements, where: (i) k is the number of (derivatives of) coefficients appearing in a product; (ii) n is the number of unique basis functions involved in the expansion. In the original example, we had $n = 3$ (for b_{i0}, b_{i1}, and b_{i2}) and $k = 1$, which confirms $\iota = 3$. In the modified example, there are two coefficients, so $k = 2$, which means $\iota = 6$.

If $\iota \geq I$ (the extent of the reduction loop), we already know that pre-evaluation will not be profitable. Intuitively, this means that we are introducing more operations than we are saving from pre-evaluating L_i. If $\iota < I$, we still need to find the number of terms ρ such that $\grave{}^{pre} = \rho \cdot \iota$. The mass matrix monomial in Figure 4.6 is characterized by the dot product of test and trial functions, so trivially $\rho = 1$. In the example in Figure 4.3, instead, we have $\rho = 3$ after a suitable factorization of basis functions. In general, therefore, ρ depends on both form and discretization. To determine this parameter, we look at the re-factorized expression (as established by Algorithm 2), and simply count the terms amenable to pre-evaluation.

4.4 Formalization

We demonstrate that the orchestration of sharing elimination and pre-evaluation performed by the transformation algorithm guarantees local optimality (Definition 10). The proof re-uses concepts and explanations provided throughout the paper, as well as the terminology introduced in Section 4.2.2.

Proposition 2. *Consider a multilinear form comprising a set of monomials M, and let Λ be the corresponding finite element integration loop nest. Let Γ be the transformation algorithm. Let X be the set of monomials that, according to Γ, need to be pre-evaluated, and let $Y = M \setminus X$. Assume that the pre-evaluation of different monomials does not result in identical tables. Then, $\Lambda' = \Gamma(\Lambda)$ is a local optimum in the sense of Definition 10 and satisfies Constraint 2.*

Proof. We first observe that the cost function θ predicts the *exact* gain/loss in monomial pre-evaluation, so X and Y can actually be constructed.

Let c_Λ denote the operation count for Λ and let $\Lambda_I \subset \Lambda$ be the subset of innermost loops (all L_k loops in Figure 4.5). We need to show that there is no other synthesis Λ''_I satisfying Constraint 2 such that $c_{\Lambda''_I} < c_{\Lambda'_I}$. This holds if and only if

1. *The coordination of pre-evaluation with sharing elimination is optimal.* This boils down to prove that
   a) *pre-evaluating any $m \in Y$ would result in $c_{\Lambda''_I} > c_{\Lambda'_I}$*
   b) *not pre-evaluating any $m \in X$ would result in $c_{\Lambda''_I} > c_{\Lambda'_I}$*
2. *Sharing elimination leads to a (at least) local optimum.*

We discuss these points separately

1. a) Let T_m represent the set of tables resulting from applying pre-evaluation to a monomial m. Consider two monomials $m_1, m_2 \in Y$ and the respective sets of pre-evaluated tables, T_{m_1} and T_{m_2}. If $T_{m_1} \cap T_{m_2} \neq \emptyset$, at least one table is assignable to the same temporary. Γ, therefore, may not be optimal, since θ only distinguishes monomials in "isolation". We neglect this scenario

(see assumptions) because of its purely pathological nature and its – with high probability – negligible impact on the operation count.

b) Let $m_1 \in X$ and $m_2 \in Y$ be two monomials sharing some generic multilinear symbols. If m_1 were carelessly pre-evaluated, there may be a potential gain in sharing elimination that is lost, potentially leading to a non-optimum. This situation is prevented by construction, because Γ exhaustively searches all possible bipartitions on order to determine an optimum which satisfies Constraint 2[4]. Recall that since the number of monomials is in practice very small, this pass can rapidly be accomplished.

2. Consider Algorithm 1. Proposition 1 ensures that there are only two ways of scheduling the multilinear operands in $P \in \mathbb{P}$: through generalized code motion (Strategy 1) or factorization of multilinear symbols (via Strategy 2). If applied, these two strategies would lead, respectively, to performing $|P|$ and $|S_P|$ multiplications at every loop iteration. Since Strategy 1 is applied if and only if $|P| < |S_P|$ and does not change the structure of the expression (it requires neither expansion nor factorization), step (1) cannot prune the optimum from the search space.

   After structuring the sharing graph G in such a way that only flop-decreasing transformations are possible, the ILP model is instantiated. At this point, proving optimality reduces to establishing the correctness of the model, which is relatively straightforward because of its simplicity. The model aims to minimize the operation count by selecting the most promising factorizations. The second set of constraints is to select all edges (i.e., all multiplications), exactly once. The first set of inequalities allows multiplications to be scheduled: once a vertex s is selected (i.e., once a symbol is decided to be factorized), all multiplications involving s can be grouped.

□

Throughout the paper we have reiterated the claim that Algorithm 3 achieves a globally optimal flop count if stronger preconditions on the

[4]Note that the problem can be seen as an instance of the well-known Knapsack problem

input variational form are satisfied. We state here these preconditions, in increasing order of complexity.

1. There is a single monomial and only a specific coefficient (e.g., the coordinates field). This is by far the simplest scenario, which requires no particular transformation at the level of the outer loops, so optimality naturally follows.

2. There is a single monomial, but multiple coefficients are present. Optimality is achieved if and only if all sub-expressions depending on coefficients are structured (see Section 4.2.2). This avoids ambiguity in factorization, which in turn guarantees that the output of step (7) in Algorithm 1 is optimal.

3. There are multiple monomials, but either at most one coefficient (e.g., the coordinates field) or multiple coefficients not inducing sharing across different monomials are present. This reduces, respectively, to cases (1) and (2) above.

4. There are multiple monomials, and coefficients are shared across monomials. Optimality is reached if and only if the coefficient-dependent sub-expressions produced by Algorithm 1 – that is, the by-product of factorizing test/trial functions from distinct monomials – preserve structure.

4.5 Code Generation

Sharing elimination and pre-evaluation, as well as the transformation algorithm, have been implemented in COFFEE, the compiler for finite element integration routines adopted in Firedrake. In this section, we briefly discuss the aspects of the compiler that are relevant for this article.

4.5.1 Expressing transformations through the COFFEE language

COFFEE implements sharing elimination and pre-evaluation by composing building block transformation operators, which we refer to as *rewrite operators*. This has several advantages. The first is extensibility. New transformations, such as sum factorization in spectral methods, could be

expressed by composing the existing operators, or with small effort building on what is already available. Second, generality: COFFEE can be seen as a lightweight, low level computer algebra system, not necessarily tied to finite element integration. Third, robustness: the same operators are exploited, and therefore tested, by different optimization pipelines. The rewrite operators, whose (Python) implementation is based on manipulation of abstract syntax trees (ASTs), comprise the COFFEE language. A non-exhaustive list of such operators includes expansion, factorization, re-association, generalized code motion.

4.5.2 Independence from form compilers

COFFEE aims to be independent of the high level form compiler. It provides an interface to build generic ASTs and only expects expressions to be in normal form (or sufficiently close to it). For example, Firedrake has transitioned from a version of the FEniCS Form Compiler `?` modified to produce ASTs rather than strings, to a newly written compiler[5], while continuing to emply COFFEE. Thus, COFFEE decouples the mathematical manipulation of a form from code optimization; or, in other words, relieves form compiler developers of the task of fine scale loop optimization of generated code.

4.5.3 Handling block-sparse tables

For several reasons, basis function tables may be block-sparse (e.g., containing zero-valued columns). For example, the FEniCS Form Compiler implements vector-valued functions by adding blocks of zero-valued columns to the corresponding tabulations; this extremely simplifies code generation (particularly, the construction of loop nests), but also affects the performance of the generated code due to the execution of "useless" flops (e.g., operations like `a + 0`). In `?`, a technique to avoid iteration over zero-valued columns based on the use of indirection arrays (e.g. `A[B[i]]`, in which `A` is a tabulated basis function and `B` a map from loop iterations to non-zero columns in A) was proposed. This technique, however, produces non-contiguous memory loads and stores, which nullify the potential benefits of vectorization. COFFEE, instead, handles block-sparse basis

[5] TSFC, the two-stage form compiler https://github.com/firedrakeproject/tsfc

function tables by restructuring loops in such a manner that low level optimization (especially vectorization) is only marginally affected. This is based on symbolic execution of the code, which enables a series of checks on array indices and loop bounds which determine the zero-valued blocks which can be skipped without affecting data alignment.

4.6 Performance Evaluation

4.6.1 Experimental setup

Experiments were run on a single core of an Intel I7-2600 (Sandy Bridge) CPU, running at 3.4GHz, 32KB L1 cache (private), 256KB L2 cache (private) and 8MB L3 cache (shared). The Intel Turbo Boost and Intel Speed Step technologies were disabled. The Intel `icc 15.2` compiler was used. The compilation flags used were `-O3, -xHost`. The compilation flag `xHost` tells the Intel compiler to generate efficient code for the underlying platform.

The Zenodo system was used to archive all packages used to perform the experiments: Firedrake **?**, PETSc **?**, petsc4py **?**, FIAT **?**, UFL **?**, FFC **?**, PyOP2 **?** and COFFEE **?**. The experiments can be reproduced using a publicly available benchmark suite **?**.

We analyze the execution time of four real-world bilinear forms of increasing complexity, which comprise the differential operators that are most common in finite element methods. In particular, we study the mass matrix ("`Mass`") and the bilinear forms arising in a Helmholtz equation ("`Helmholtz`"), in an elastic model ("`Elasticity`"), and in a hyperelastic model ("`Hyperelasticity`"). The complete specification of these forms is made publicly available[6].

We evaluate the speed-ups achieved by a wide variety of transformation systems over the "original" code produced by the FEniCS Form Compiler (i.e., no optimizations applied). We analyze the following transformation systems:

quad Optimized quadrature mode. Work presented in **?**, implemented in in the FEniCS Form Compiler.

[6] https://github.com/firedrakeproject/firedrake-bench/blob/experiments/forms/firedrake_forms.py

tens Tensor contraction mode. Work presented in **?**, implemented in the FEniCS Form Compiler.

auto Automatic choice between `tens` and `quad` driven by heuristic (detailed in Logg et al. [2012] and summarized in Section **??**). Implemented in the FEniCS Form Compiler.

ufls UFLACS, a novel back-end for the FEniCS Form Compiler whose primary goals are improved code generation and execution times.

cfO1 Generalized loop-invariant code motion. Work presented in **?**, implemented in COFFEE.

cfO2 Optimal loop nest synthesis with handling of block-sparse tables. Work presented in this article, implemented in COFFEE.

The values that we report are the average of three runs with "warm cache"; that is, with all kernels retrieved directly from the Firedrake's cache, so code generation and compilation times are not counted. The timing includes however the cost of both local assembly and matrix insertion, with the latter minimized through the choice of a mesh (details below) small enough to fit the L3 cache of the CPU.

For a fair comparison, small patches were written to the make `quad`, `tens`, and `ufls` compatible with Firedrake. By executing all simulations in Firedrake, we guarantee that both matrix insertion and mesh iteration have a fixed cost, independent of the transformation system employed. The patches adjust the data storage layout to what Firedrake expects (e.g., by generating an array of pointers instead of a pointer to pointers, by replacing flattened arrays with bi-dimensional ones).

For Constraint 2, discussed in Section 4.2.4, we set $T_H = \text{size}(\text{L2})$; that is, the size of the processor L2 cache (the last level of private cache). When the threshold had an impact on the transformation process, the experiments were repeated with $T_H = \text{size}(\text{L3})$. The results are documented later, individually for each problem.

Following the methodology adopted in **?**, we vary the following parameters:

- the polynomial degree of test, trial, and coefficient (or "pre-multiplying") functions, $q \in \{1, 2, 3, 4\}$

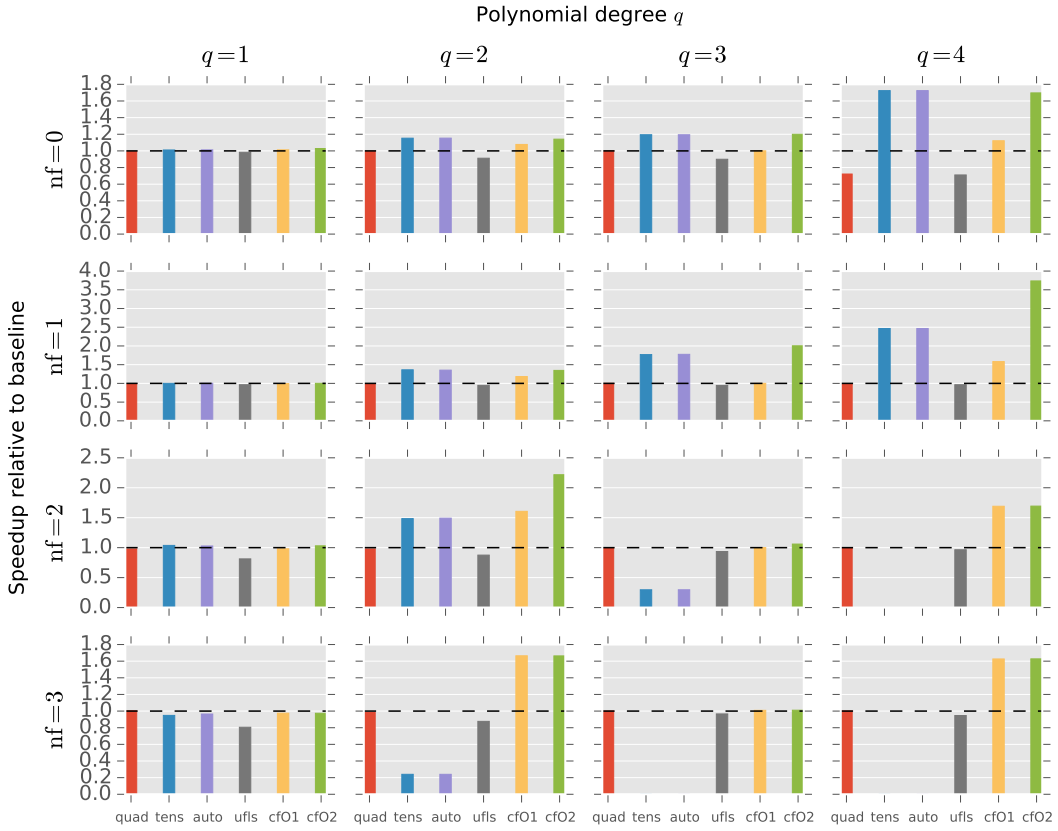


Figure 4.7: Performance evaluation for the *mass* matrix. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

- the number of coefficient functions nf $\in \{0, 1, 2, 3\}$

While constants of our study are

- the space of test, trial, and coefficient functions: Lagrange
- the mesh: tetrahedral with a total of 4374 elements
- exact numerical quadrature (we employ the same scheme used in **?**, based on the Gauss-Legendre-Jacobi rule)

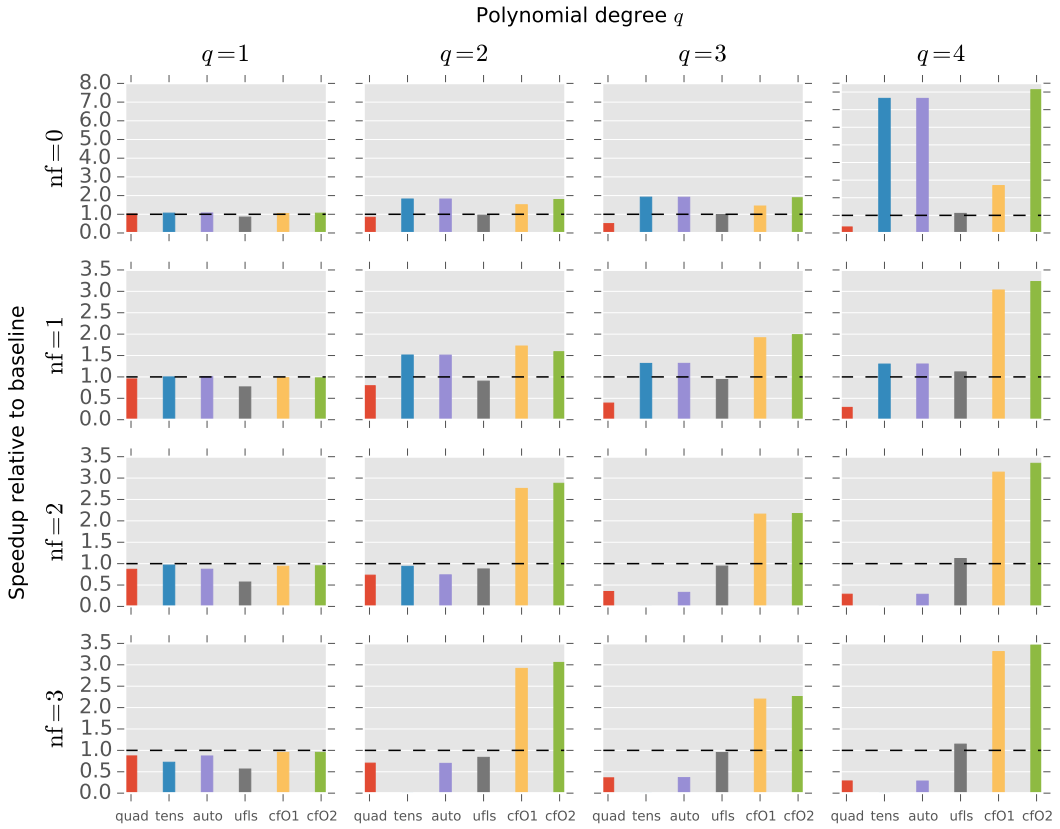


Figure 4.8: Performance evaluation for the bilinear form of a *Helmholtz* equation. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

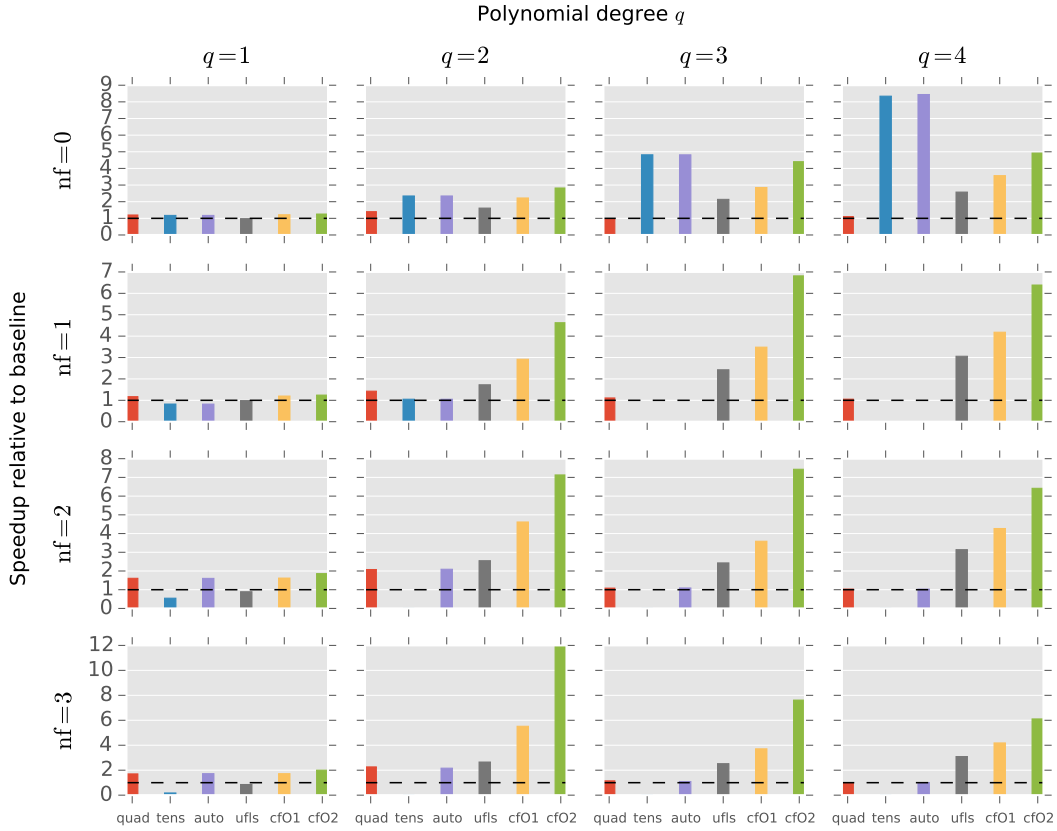


Figure 4.9: Performance evaluation for the bilinear form arising in an *elastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

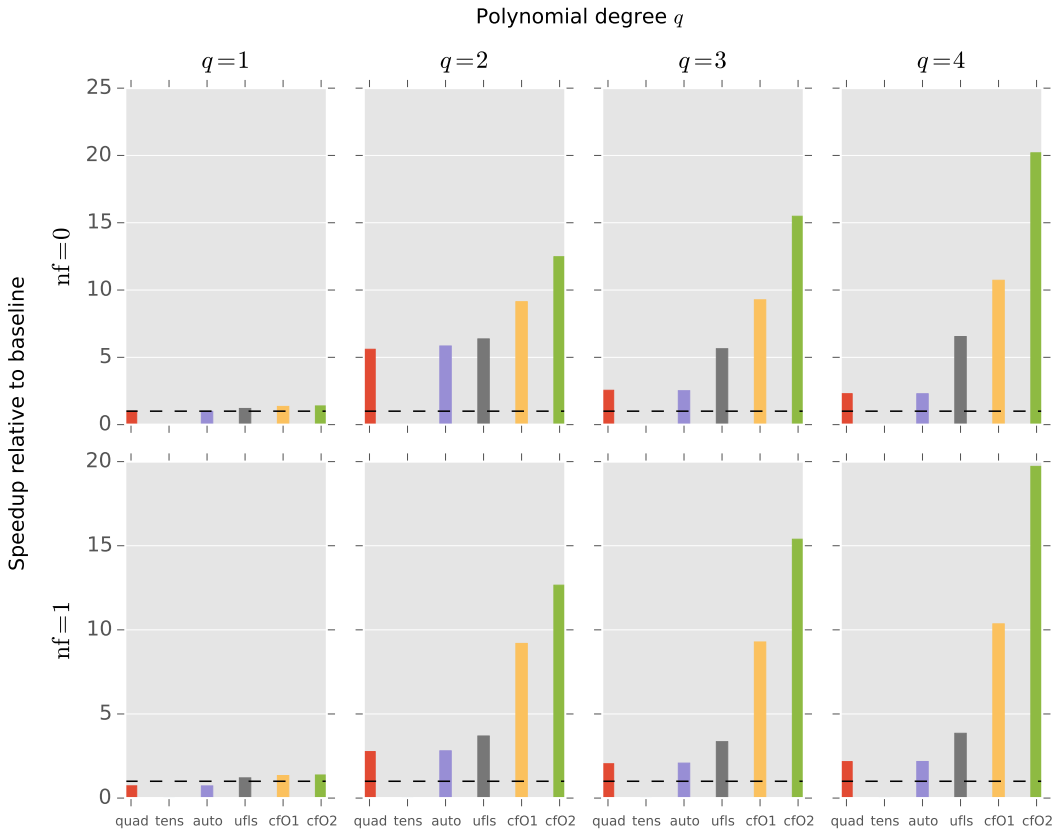


Figure 4.10: Performance evaluation for the bilinear form arising in a *hyperelastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

4.6.2 Performance results

We report the results of our experiments in Figures 4.7, 4.8, 4.9, and 4.10 as three-dimensional plots. The axes represent q, nf, and code transformation system. We show one subplot for each problem instance $\langle \text{form}, \text{nf}, q \rangle$, with the code transformation system varying within each subplot. The best variant for each problem instance is given by the tallest bar, which indicates the maximum speed-up over non-transformed code. We note that if a bar or a subplot are missing, then the form compiler failed to generate code because it either exceeded the system memory limit or was otherwise unable to handle the form.

The rest of the section is organized as follows: we first provide insights into the general outcome of the experimentation; we then comment on the impact of a fundamental low-level optimization, namely autovectorization; finally, we motivate, for each form, the performance results obtained.

High level view Our transformation strategy does not always guarantee minimum execution time. In particular, about 5% of the test cases (3 out of 56, without counting marginal differences) show that `cf02` was not optimal in terms of runtime. The most significant of such test cases is the elastic model with $[q = 4,\ \text{nf} = 0]$. There are two reasons for this. First, low level optimization can have a significant impact on the actual performance. For example, the aggressive loop unrolling in `tens` eliminates operations on zeros and reduces the working set size by not storing entire temporaries; on the other hand, preserving the loop structure can maximize the chances of autovectorization. Second, the transformation strategy adopted when T_H is exceeded plays a key role, as we will later elaborate.

Autovectorization We chose the mesh dimension and the function spaces such that the inner loop sizes would always be a multiple of the machine vector length. This ensured autovectorization in the majority of code variants[7]. The biggest exception is `quad`, due to the presence of indirection

[7] We verified the vectorization of inner loops by looking at both compiler reports and assembly code.

arrays in the generated code. In `tens`, loop nests are fully unrolled, so the standard loop vectorization is not feasible; the compiler reports suggest, however, that block vectorization **?** is often triggered. In `ufls`, `cf01`, and `cf02` the iteration spaces have identical structure, with loop vectorization being regularly applied.

Mass matrix We start with the simplest of the bilinear forms investigated, the mass matrix. Results are in Figure 4.7. We first notice that the lack of improvements when $q = 1$ is due to the fact that matrix insertion outweighs local assembly. For $q \geq 2$, `cf02` generally shows the highest speed-ups. It is worth noting why `auto` does not always select the fastest implementation: `auto` always opts for `tens`, while for nf ≥ 2 `quad` tends to be preferable. On the other hand, `cf02` always makes the optimal decision about whether to apply pre-evaluation or not. Surprisingly, despite the simplicity of the form, the performance of the various code generation systems can differ significantly.

Helmholtz As in the case of Mass matrix, when $q = 1$ the matrix insertion phase is dominant. For $q \geq 2$, the general trend is that `cf02` outperforms the competitors. In particular:

nf $= 0$ pre-evaluation makes `cf02` notably faster than `cf01`, especially for high values of q; `auto` correctly selects `tens`, which is comparable to `cf02`.

nf $= 1$ `auto` picks `tens`; the choice is however sub-optimal when $q = 3$ and $q = 4$. This can indirectly be inferred from the large gap between `cf02` and `tens`/`auto`: `cf02` applies sharing elimination, but it correctly avoids pre-evaluation because of the excessive expansion cost.

nf $= 2$ **and** nf $= 3$ `auto` reverts to `quad`, which would theoretically be the right choice (the flop count is much lower than in `tens`); however, the generated code suffers from the presence of indirection arrays, which break autovectorization and "traditional" code motion.

The slow-downs (or marginal improvements) seen in a small number of cases exhibited by `ufls` can be attributed to the presence of sharing in the

generated code.

An interesting experiment we additionally performed was relaxing the memory threshold by setting $T_H = \text{size(L3)}$. We found that this makes `cf02` generally slower for nf ≥ 2, with a maximum slow-down of 2.16× with $\langle \text{nf} = 2, q = 2 \rangle$. This effect could be worse when running in parallel, since the L3 cache is shared and different threads would end up competing for the same resource.

Elasticity The results for the elastic model are displayed in Figure 4.9. The main observation is that `cf02` never triggers pre-evaluation, although in some occasions it should. To clarify this, consider the test case $\langle \text{nf} = 0,\ q = 2 \rangle$, in which `tens/auto` show a considerable speed-up over `cf02`. `cf02` finds pre-evaluation profitable in terms of operation count, although it is eventually not applied to avoid exceeding T_H. However, running the same experiments with $T_H = \text{size(L3)}$ resulted in a dramatic improvement, even higher than that obtained by `tens`. The reason is that, despite exceeding T_H by roughly 40%, the saving in operation count is so large (5× in this specific problem) that pre-evaluation would in practice be the winning choice. This suggests that our objective function should be improved to handle the cases in which there is a significant gap between potential cache misses and reduction in operation count.

We also note that:

- the differences between `cf02` and `cf01` are due to the perfect sharing elimination and the zero-valued blocks avoidance technique presented in Section 4.5.3.
- when nf $= 1$, `auto` prefers `tens` over `quad`, which leads to suboptimal operation counts and execution times.
- `ufls` often results in better execution times than `quad` and `tens`. This is due to multiple factors, including avoidance of indirection arrays, preservation of loop structure, and a more effective code motion strategy.

Hyperelasticity In the experiments on the hyperelastic model, shown in Figure 4.10, `cf02` exhibits the largest gains out of all problem instances

considered in this paper. This is a positive result, since it indicates that our transformation algorithm scales well with form complexity. The fact that all code transformation systems (apart from `tens`) show quite significant speed-ups suggests two points. First, the baseline is highly inefficient. With forms as complex as in the hyperelastic model, a trivial translation of integration routines into code should always be avoided as even the best general-purpose compiler available (the Intel compiler on an Intel platform at maximum optimization level) fails to exploit the structure inherent in the expressions. Second, the strategy for removing spatial and temporal sharing has a tremendous impact. Sharing elimination as performed by `cf02` ensures a critical reduction in operation count, which becomes particularly pronounced for higher values of q.

4.7 Conclusions

We have developed a theory for the optimization of finite element integration loop nests. The article details the domain properties which are exploited by our approach (e.g., linearity) and how these translate to transformations at the level of loop nests. All of the algorithms shown in this paper have been implemented in COFFEE, a compiler publicly available fully integrated with the Firedrake framework. The correctness of the transformation algorithm was discussed. The performance results achieved suggest the effectiveness of our methodology.

4.8 Limitations and future work

We have defined sharing elimination and pre-evaluation as high level transformations on top of a specific set of rewrite operators, such as code motion and factorization, and we have used them to construct the transformation space. There are three main limitations in this process. First, we do not have a systematic strategy to optimize sub-expressions which are independent of linear loops. Although we have a mechanism to determine how much computation should be hoisted to the level of the integration (reduction) loop, it is not clear how to effectively improve the heuristics used at step (6) in Algorithm 1. Second, lower operation counts may be found by exploiting domain-specific properties, such as redundancies in

basis functions; this aspect is completely neglected in this article. Third, with Constraint 1 we have limited the applicability of code motion. This constraint was essential given the complexity of the problem tackled.

Another issue raised by the experimentation concerns selecting a proper threshold for Constraint 2. To solve this problem would require a more sophisticated cost model, which is an interesting question deserving further research.

We also identify two additional possible research directions: a complete classification of forms for which a global optimum is achieved; and a generalization of the methodology to other classes of loop nests, for instance those arising in spectral element methods.

...

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 unite-stride memory accesses are present.

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

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

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

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

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

- Extensive experimentation using a set of real-world forms commonly arising in finite element methods.

- A discussion concerning the generality of the transformations and their applicability to different domains.

5.2 Low-level Optimization

5.2.1 Padding and Data Alignment

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

Data alignment is enforced in two steps. Firstly, all arrays (but the element matrix, for reasons discussed shortly) are padded by rounding the innermost dimension to the nearest multiple of `VL`. For instance, assume the original size of a basis function array is 3×3 and `VL` $= 4$ (e.g. AVX processor, with 32-byte long vector registers and 8-byte double-precision floats). In this case, a padded version of the array will have size 3×4. Secondly, their base address is enforced to multiples of `VL` by means of special attributes. The compiler is explicitly told about data alignment using suitable pragmas; for example, in the case of the Intel compiler, the annotation `#pragma vector aligned` is added before the loop (as shown in later figures) to inform that all of the memory accesses in the loop body will be properly aligned. This allows the compiler to issue aligned load and store instructions, which are notably faster than unaligned ones.

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

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

Listing 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 2, 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 `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       __m256d 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

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

where B is the set of all basis functions or temporary variables accessed in the kernel, whereas B' and B'' are generic problem-dependent subsets. Regardless of the specific input problem, by abstracting from the presence of all variables independent of both `j` and `k`, the element matrix computation is always reducible to this form. Figure 5.1 illustrates how we can evaluate 16 entries ($j,k = 0,...,3$) of the element matrix using just 2 vector registers, which represent a 4×4 tile, assuming $|B'| = |B''| = 1$. Values in a register are shuffled each time a product is performed. Standard compiler auto-vectorization for both GNU and Intel compilers, instead, executes 4 broadcast operations (i.e., "splat" of a value over all of the register locations) along the outer dimension to perform the calculation. In addition to incurring a larger number of cache accesses, it needs to keep between $f = 1$ and $f = 3$ extra registers to perform the same 16 evaluations when unroll-and-jam is used, with f being the unroll-and-jam factor.

The storage layout of A, however, is incorrect after the application of this outer-product-based vectorization (*op-vect*, in the following). It can be efficiently restored with a sequence of vector shuffles following the pattern highlighted in Figure 5.2, executed once outside of the `ijk` loop nest. The pseudo-code for the weighted Laplace assembly kernel using *op-vect* is shown in Listing 3.

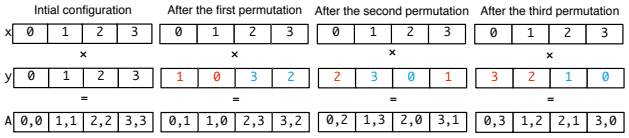


Figure 5.1: Outer-product vectorization by permuting values in a vector register.

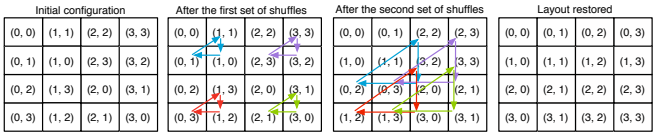


Figure 5.2: Restoring the storage layout after *op-vect*. The figure shows how 4×4 elements in the top-left block of the element matrix A can be moved to their correct positions. Each rotation, represented by a group of three same-colored arrows, is implemented by a single shuffle intrinsic.

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 **??**. 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 2), where a larger number of basis function arrays ($X1$, $X2$, ...) and constants ($F0$, $F1$, ..., $K0$, $K1$,...) are generated.

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

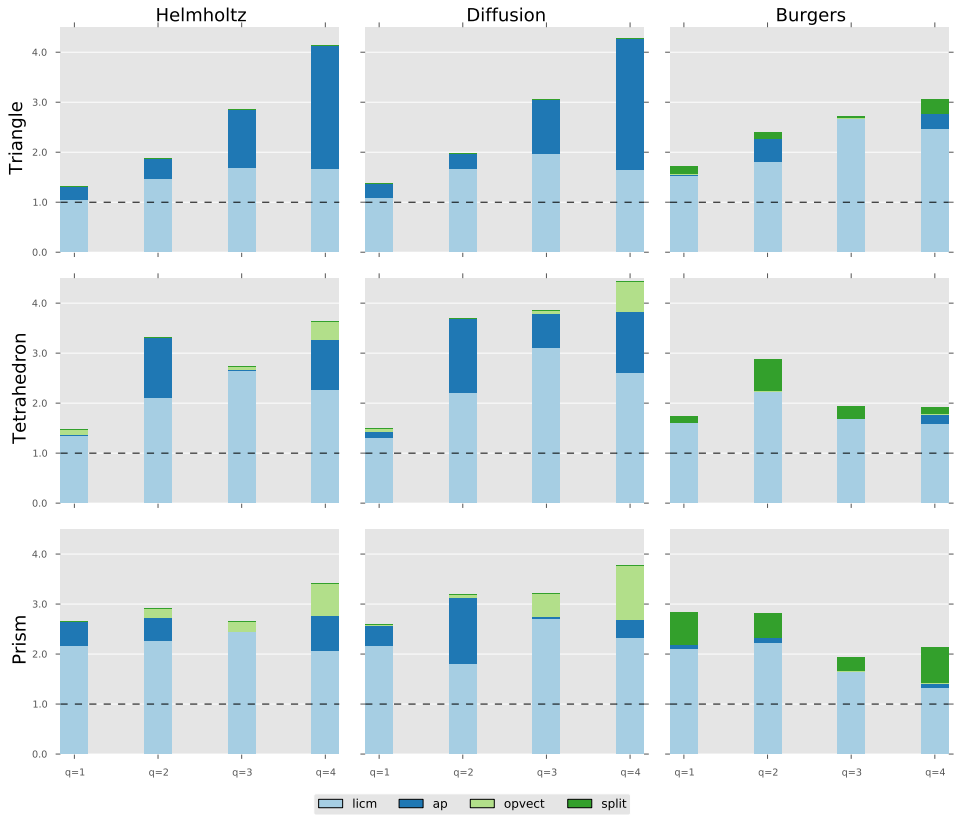


Figure 5.3: Performance improvement due to generalized loop-invariant code motion (*licm*), data alignment and padding (*ap*), outer-product vectorization (*op-vect*), and expression splitting (*split*) over the original non-optimized code. In each plot, the horizontal axis reports speed ups, whereas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

5.3.2 Impact of Transformations

Experiments were run on a single core of an Intel architecture, a Sandy Bridge I7-2600 CPU running at 3.4 GHz, with 32KB of L1 cache and 256KB of L2 cache). The `icc 14.1` compiler was used. On the Sandy Bridge, the compilation flags used were `-O2` and `-xAVX` for auto-vectorization (other optimization levels were tried, but they generally resulted in higher execution times).

The speed-ups achieved by applying the transformations on top of the original assembly kernel code are shown in Figure 5.3. This figure is a

three-dimensional plot: element shape and equation vary along the outermost axes, whereas q varies within each sub-plot. In the next sections, we will refer to this figure and elaborate on the impact of the individual transformations. We shorten generalized loop-invariant code motion as *licm*; padding and data alignment as *ap*; outer-product vectorization as *op-vect*; expression splitting as *split*.

Impact of Generalized Loop-invariant Code Motion

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

The gain tends to grow with the computational cost of the kernels: bigger loop nests (i.e., larger element shapes and polynomial orders) usually benefit from the reduction in redundant computation, even though extra memory for the temporary arrays is required. Some discrepancies to this trend are due to a less effective auto-vectorization. For instance, on the Sandy Bridge, the improvement at $q = 3$ is larger than that at $q = 4$ because, in the latter case, the size of the innermost loop is not a multiple of the vector length, and a remainder scalar loop is introduced at compile time. Since the loop nest is small, the cost of executing the extra scalar iterations can have a significant impact.

Impact of Padding and Data Alignment

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

Impact of Vector-register Tiling

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

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

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

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

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

Impact of Expression Splitting

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

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

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

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

The *split* transformation was also tried in combination with *op-vect* (*split-op-vect*). Despite improvements up to 1.22×, *split-op-vect* never outperforms *split*. This is motivated by two factors: for small split factors, such as 1 and 2, the data space to be tiled is still too big, and register spilling affects run-time; for higher ones, sub-expressions become so small that, as explained in Section 5.3.2, extensive unrolling already allows to achieve a certain degree of register re-use.

5.4 Experience with Traditional Compiler Optimizations

5.4.1 Loop Interchange

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

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

5.4.2 Loop Unroll

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

We first observe that manual full (or extensive) unrolling is unlikely to be effective for two reasons. Firstly, the `ijk` loop nest would need to be small enough such that the unrolled instructions do not exceed the instruction cache, which is rarely the case: it is true that in a local assembly kernel the minimum size of the `ijk` loop nest is $3\times3\times3$ (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; r++) {
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 is another computational domains, and to what extent they would be helpful for improving the full application performance. To answer these questions, we first need to go back to the origins of our study. The starting point of our work was the mathematical formulation of a finite element operator, expressible as follows

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

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

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

  1. In Equation (5.2), the summations correspond to reduction loops, whereas loops over indices i and j are fully parallel. Throughout the paper we assumed that a kernel will be executed by a single thread, which is likely to be the best strategy for standard multi-core CPUs. On the other hand, we note that for cer-

tain architectures (for example GPUs) this could be prohibitive due to memory requirements. Intra-kernel parallelization is one possible solution: a domain-specific compiler could map mathematical quantifiers and operators to different parallelization schemes and generate distinct variants of multi-threaded kernel code. Based on our experience, we believe this is the right approach to achieve performance portability.

2. The various sub-expressions in β only depend on (i.e. iterate along) a subset of the enclosing loops. In addition, some of these sub-expressions might reduce to the same values as iterating along certain iteration spaces. This code structure motivated the generalized loop-invariant code motion technique. The intuition is that whenever sub-expressions invariant with respect to different sets of affine loops can be identified, the question of whether, where and how to hoist them, while minimizing redundant computation, arises. Pre-computation of invariant terms also increases memory requirements due to the need for temporary arrays, so it is possible that for certain architectures the transformation could actually cause slowdowns (e.g. whenever the available per-core memory is small).

3. Associative arithmetic operators are the prerequisite for expression splitting. In essence, this transformation concerns resource-aware execution. In our context, expression splitting has successfully been applied to improve register pressure. However, the underlying idea of re-scheduling (re-associating) operations to optimize for some generic parameters is far more general. It could be used, for example, as a starting point to perform kernel fission; that is, splitting a kernel into multiple parts, each part characterized by less stringent memory requirements (a variant of this idea for non-affine loops in unstructured mesh applications has been adopted in [Bertolli et al., 2013]). In Equation (5.2), for instance, not only can any of the functions α, β and γ be split (assuming they include associative operators), but α could be completely extracted and evaluated in a separate kernel. This would reduce the working set size of each

of the kernel functions, an option which is particularly attractive for many-core architectures in which the available per-core memory is much smaller than that in traditional CPUs.

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

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

processor. By demonstrating the effectiveness of multi-loop vectorization in a real scenario, our research represents an incentive for studying this technique in a broader and systematic way.

5.7 Conclusion

In this chapter, we have presented the study and systematic performance evaluation of a class of composable cross-loop optimizations for improving arithmetic intensity in finite element local assembly kernels. In the context of automated code generation for finite element local assembly, COFFEE is the first compiler capable of introducing low-level optimizations to simultaneously maximize register locality and SIMD vectorization. Assembly kernels have particular characteristics. Their iteration space is usually very small, with the size depending on aspects like the degree of accuracy one wants to reach (polynomial order of the method) and the mesh discretization employed. The data space, in terms of number of arrays and scalars required to evaluate the element matrix, grows proportionally with the complexity of the finite element problem. The various optimizations overcome limitations of current vendor and research compilers. The exploitation of domain knowledge allows some of them to be particularly effective, as demonstrated by our experiments on a state-of-the-art Intel platform. The generality and the applicability of the proposed code transformations to other domains has also been discussed.

Chapter 6

COFFEE: a Compiler for Fast Expression Evaluation

6.1 Overview

Sharing elimination and pre-evaluation, which we presented in Chapter 4, as well as the low level optimizations discussed in Chapter 5, have been implemented in COFFEE[1], a mature, platform-agnostic compiler. COFFEE has fully been integrated with Firedrake, the framework based on the finite element method introduced in Section **??**. The code, which comprises more than 5000 lines of Python, is available at [Luporini, 2014a].

Firedrake users employ the Unified Form Language to express problems in a notation resembling mathematical equations. At run-time, the high-level specification is translated by a form compiler, the Two-Stage Form Compiler (TSFC) **?**, into one or more abstract syntax trees (ASTs) representing assembly kernels. ASTs are then passed to COFFEE for optimization. The output of COFFEE, C code, is eventually provided to PyOP2 [Markall et al., 2013], where just-in-time compilation and execution over the discretized domain take place. The flow and the compiler structure are outlined in Figure 6.1.

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

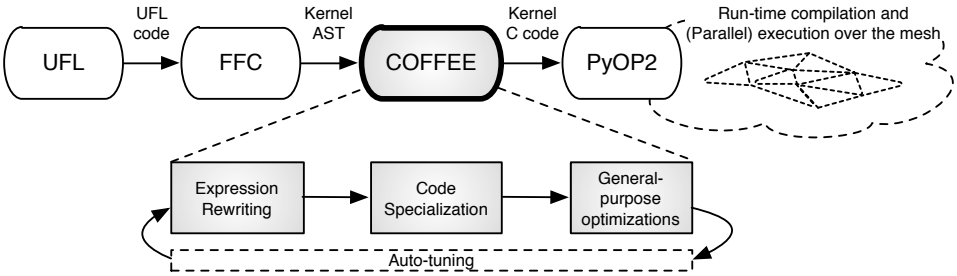


Figure 6.1: High-level view of Firedrake. COFFEE is at the core, receiving ASTs from a modified version of the FEniCS Form Compiler and producing optimized C code kernels.

6.2 The Compilation Pipeline

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

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

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

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

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

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

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

Phase 1: analysis During the analysis phase, an AST is visited and several kinds of information are collected. In particular, COFFEE searches for expression rewriting candidates. These are represented by special nodes in the AST, which we refer to as "expression nodes". In plain C, we could think of an expression node as a statement preceded by a directive such as `#pragma coffee expression`; the purpose of the directive would be to trigger COFFEE's `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 1).

03 Algorithm 3, which coordinates sharing elimination and pre-evaluation, is applied. This is followed by handling block-sparse tables, and finally by padding and data alignment.

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

6.3 Plugging COFFEE into Firedrake

6.3.1 Abstract Syntax Trees

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

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

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

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

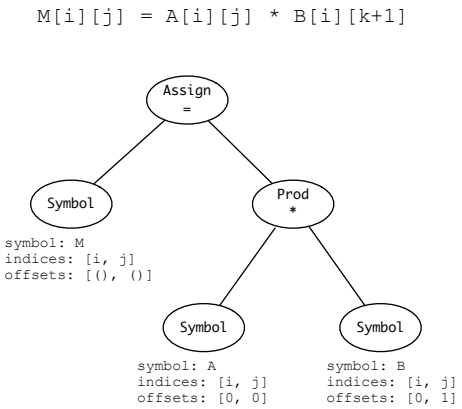


Figure 6.2: AST representation of a C assignment in COFFEE.

entry r in the rank, there is a corresponding entry $<scale, stride>$ in the offset. Rank and offset are used as in Figure 6.2 to access specific memory locations. By clearly identifying rank and offset of a `Symbol` – rather than storing a generic expression – the complexity of the data dependency analysis required by the rewrite operators is greatly reduced. The underlying assumption, however, is that all symbols in the kernel (at least those relevant for optimization) have access functions (see Section 2.5) that are affine in the loop indices. As motivated in Chapter 4, this is definitely the case for the class of kernels in which we are interested.

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

6.3.2 Integration with Form Compilers

So far, COFFEE has been integrated with two form compilers: the FEniCS Form Compiler (FFC) and the Two-Stage Form Compiler (TSFC)[3]. These

[3]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 4.1, have been provided throughout the thesis). At the price of some extra memory for storing temporaries, all lifted terms are now amenable to auto-vectorization.

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

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

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

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

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

6.5 Features of the Implementation

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

6.5.1 Tree Visitor Pattern

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

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

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

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

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

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

6.5.2 Flexible Code Motion

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

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

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

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

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

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

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

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

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

6.5.3 Tracking Data Dependency

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

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

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

6.5.4 Minimizing Temporaries

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

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

Chapter 7

Conclusions

...

Bibliography

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

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

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

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.

Federico Bassetti, Kei Davis, and Daniel J. Quinlan. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, ISCOPE '98, pages 107–118, London, UK, UK, 1998. Springer-

Verlag. ISBN 3-540-65387-2. URL http://dl.acm.org/citation.cfm?id=646894.709707.

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

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

Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron, and Nicolas Vasilache. Tiling and optimizing time-iterated computations on periodic domains. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 39–50, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. doi: 10.1145/2628071.2628106. URL http://doi.acm.org/10.1145/2628071.2628106.

Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled qr factorization for multicore architectures. In *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics*, PPAM'07, pages 639–648, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-68105-1, 978-3-540-68105-2. URL http://dl.acm.org/citation.cfm?id=1786194.1786268.

Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, January 2009. ISSN 0167-8191. doi: 10.1016/j.parco.2008.10.002. URL http://dx.doi.org/10.1016/j.parco.2008.10.002.

Li Chen, Zhao-Qing Zhang, and Xiao-Bing Feng. Redundant computation partition on distributed-memory systems. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 252–260, Oct 2002. doi: 10.1109/ICAPP.2002.1173583.

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

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

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

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

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

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

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.

Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012. doi: 10.1142/S0129626412500107. URL http://www.worldscientific.com/doi/abs/10.1142/S0129626412500107.

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.

Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304619. URL http://doi.acm.org/10.1145/2304576.2304619.

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

Andreas Klöckner. Loo.py: Transformation-based code generation for gpus and cpus. In *Proceedings of ACM SIGPLAN International Workshop*

on Libraries, Languages, and Compilers for Array Programming, ARRAY'14, pages 82:82–82:87, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8. doi: 10.1145/2627373.2627387. URL http://doi.acm.org/10.1145/2627373.2627387.

Christopher D Krieger. *Generalized full sparse tiling of loop chains*. PhD thesis, Colorado State University, 2013.

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

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

Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235–244, June 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250761. URL http://doi.acm.org/10.1145/1273442.1250761.

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

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

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

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

Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-498-0. doi: 10.1145/1542275.1542313. URL http://doi.acm.org/10.1145/1542275.1542313.

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

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.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176. URL http://doi.acm.org/10.1145/2491956.2462176.

Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In

Proc. Intl. Conf. on High Perf. Comp., Net., Sto. & Anal., pages 72:1–72:11, 2012. ISBN 978-1-4673-0804-5.

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

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

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

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

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

Michelle Mills Strout. Compilers for regular and irregular stencils: Some shared problems and solutions. In *Proceedings of Workshop on Optimizing Stencil Computations (WOSC)*, October 27, 2013.

Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular

Gauss-Seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, July 2002.

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

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

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

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

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.

Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259044. URL http://doi.acm.org/10.1145/2259016.2259044.

Gerhard Zumbusch. Vectorized higher order finite difference kernels. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing*, PARA'12, pages 343–357, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-36802-8. doi: 10.1007/978-3-642-36803-5_25. URL http://dx.doi.org/10.1007/978-3-642-36803-5_25.