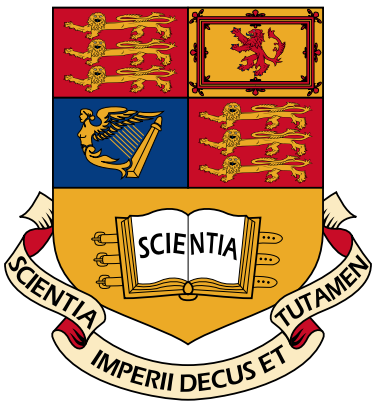
Imperial College London
Department of Computing

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 Brenner and Scott [2007], 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 magic of such a black box is clearly out of the scope of this chapter. We rather limit ourselves to review the mathematical and 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 by Rathgeber [2014], Logg et al. [2012] and Olgaard and Wells [2010]. For a complete treatment of the subject, the reader is invited to refer to Brenner and Scott [2007].

2.1.1 Weak Formulation

In a *boundary value problem*, the solution of a (partial) differential equation subjected to a set of *boundary conditions* is sought. The boundary conditions specify the behaviour of the unknown at the domain boundary. We

consider the *variational* or *weak* formulation of one such problem

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

where u is the sought solution, a a bilinear form and L a linear form. The term "variational" stems from the fact that the function v can vary arbitrarily. The reader unfamiliar with the theory of Hilbert spaces may find this formulation unusual. Informally, we can think of V as a "very nice" space, in which functions have desirable properties. The underlying idea of the variational formulation consists of "transferring" certain requirements on the unknown (e.g., differentiability) to v.

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

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

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

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

This allows us to rewrite (2.2) as:

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

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

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

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 is again invited to refer to Brenner and Scott [2007].

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*. Formally, a finite element is 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 imposing 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

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

where δ_{ij} is the Kronecker delta, must be satisfied. 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 The following example is extracted from Logg et al. [2012]. Consider a triangular cell K and let $\mathcal{P}_K$ be the space of polynomials of order 1 on K. Let $\mathcal{L}_K$ 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:

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

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 $U_h = \text{span}\{\psi_j\}_{j=1}^N$ 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 Logg et al. [2012] for a comprehensive description of this step.

One of the crucial aspects of an FEM is that global function spaces are

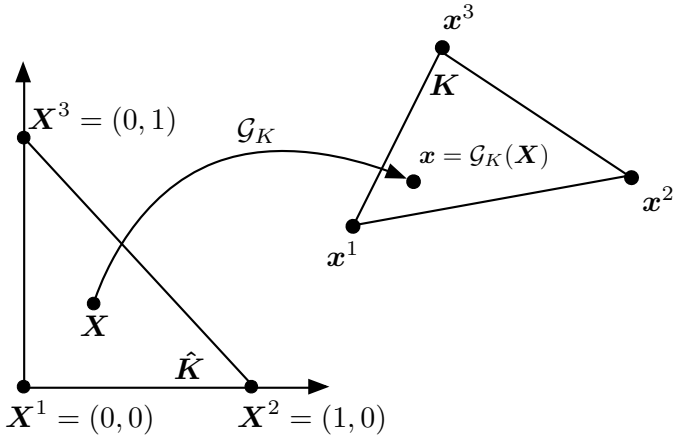


Figure 2.1: Affine mapping from the reference element $\hat{K}$ to an element K.

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 2.1.

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 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 in (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. Since the operator is linear, 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 $i \in \mathcal{I}_K$, the index set on A_i^K. That is, $\mathcal{I}_K = \{(1,1), ..., (n_U, n_V)\}$, 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

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

This simply maps a pair of local degrees of freedom to a pair of global degrees of freedom. Let $\mathcal{T}$ 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 formulate the computation of A as the sum of local contributions from the elements 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 corresponding inverse map. 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 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 associated with the weak variational form of the equation is:

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

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.

Monomials

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. We refer to such integrals as the *monomials* of the form.

Quadrature Mode

A_{ij}^K is numerically evaluated by means of a quadrature scheme. A quadrature scheme evaluates an integral at a set of *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 by using the assembly procedure detailed 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 inverse map $\mathcal{G}_K^{-1}$, which allows us to write ϕ_i^K as $\phi_i^K = \Phi_i \circ \mathcal{G}_K^{-1}$. 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. Exploiting linearity, associativity and distributivity of the involved mathematical

operators, we can rewrite (2.21) as

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

A generalization of this transformation has been proposed in Kirby and Logg [2006]. By 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

$$A_{ij}^K = \sum_{\alpha} A_{i_1 i_2 \alpha}^0 G_K^\alpha \quad (2.23)$$

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 [Kirby and Logg, 2006], 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 introduce a novel code generation system that goes beyond the dichotomy between quadrature and tensor modes.

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);
      }
    }
  }
}
```

Code Examples

A possible C implementation of (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 summation

over α_3 for expressing w is implemented by the loop `r`. The summations over the spatial dimensions α_1, α_2 and β have 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 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];
      }
    }
  }
}
```

The evaluation of integrals becomes more computationally expensive if the complexity of the variational form grows, in terms of number of coefficients and differential or algebraic operators involved. It is not pathological a scenario in which the computation of the 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 assembly 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 choice of finite elements.

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 behind the 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. 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

The performance optimizations studied in this thesis target different layers of abstraction. In this section, we dive into such abstractions and review the state-of-the-art on established software. This will provide the foundation for understanding 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 finite element methods 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 [Rathgeber et al., 2015] projects. In these frameworks, the weak variational formulation of a problem is expressed at high level by means of a domain-specific language. The mathematical specification is manipulated by a form compiler that generates a representation of the assembly operators. Such representation may first be transformed for performance optimization and, subsequently, translated into C code, compiled and executed. This entire process occurs at run-time: both FEniCS and Firedrake were indeed written in Python to simplify the analysis of the top-level domain-specific language. When the operators are assembled, a linear system needs be solved. For this, the PETSc library [Balay et al., 2015] is employed. In the following, we expand on the components 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* [Alnæs et al., 2014]. UFL is unrelated to meshes, function spaces, and solvers. It only concerns with the variational formulation of a problem. It provides different kinds of operators, including differential operators (e.g., `grad` for taking the gradient of a function, `inner` to compute inner products) and algebraic operators (e.g., `transpose`, `inverse`), as well as elementary functions (e.g., `abs` for the absolute value, `sqrt` for the square root). UFL starts analyzing the input form to collect some useful information; it then applies some preliminary transformations, such as automatic differentiation; eventually, it emits a representation suitable for the underlying compiler.

The 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

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
```

family represents the element type. UFL supports several families, including the traditional *Lagrange* and *Discontinuous Galerkin* elements as well as mixed elements such as *H(div)* and *H(curl)*. This allows solving problems with different requirements on the continuity of the functions, as thoroughly described in Logg et al. [2012]. The *cell* represents the shape of the reference element: possible values include *triangle*, *quadrilateral* and *tetrahedron*. The *polynomial degree* 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*.

UFL pioneered the development of scientific methods through high level mathematical notation. It is therefore worth appreciating the depth of the language and the power of the underlying transformation system, although a deep knowledge is unnecessary for the rest of this thesis.

Form Compilers

The transformed UFL (e.g., after automatic differentiation has been applied) is passed to a form compiler, which now has to construct a representation of the assembly operators. The *FEniCS Form Compiler*, or FFC, was originally used by Firedrake for this task. FFC, which supports the quadrature and tensor contraction modes illustrated in Section 2.1.5, was adapted from FEniCS to emit abstract syntax trees (ASTs) instead of C code. More recently, FFC has been supplanted by 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 ASTs in a lower-level compiler, COFFEE, whose structure will be 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 test or trial functions) exploitable for later optimization.

- COFFEE applies code transformations for improving the performance of the operators returned by TSFC.

The conception and the design of the COFFEE layer is one of the main contributions of this thesis. Chapters 4, 5 and 6 are entirely devoted to this topic.

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*, a domain specific language embedded in Python relying on just-in-time compilation and execution [Markall et al., 2013].

Several kinds of kernels may need be executed in a Firedrake program. Computationally expensive kernels arise from the assembly operators presented in Section 2.1.4. Other kernels are required for applying boundary conditions as well as for 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 receive contributions from 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 a linear system as (2.5).

The PyOP2 abstraction implements another separation of concerns. The kernels, which encode the numerics of an FEM, are produced at higher layers through the Firedrake's language and form compiler, while the parallel application of kernels over the mesh is entirely handled 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 represents meshes by means of 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). Sets and maps are constructed at a higher layer, in particular by Firedrake with the help of an additional software module, PETSc's *DMPlex* [Lange et al., 2015].

DMPlex is a data management abstraction representing unstructured mesh data through direct acyclic graphs. It relieves PyOP2 from the duty of handling complex operations such as partitioning for parallel execution and reordering for efficient memory accesses. The abstraction used in DMPlex is also so flexible to enable a number of optimizations. For instance, the fact that partition boundaries can be made arbitrarily deep facilitates the implementation of execution schemes based on communication-computation overlap, as well as the introduction of low level transformations such as loop fusion (a contribution of this thesis, Chapter 3).

Solving systems of equations

One of the key ideas behind the success of Firedrake is that of relying on already available software to implement specific components of an FEM. This philosophy is also applied to the resolution of linear systems, for which the *Portable, Extensible Toolkit for Scientific Computation* library [Balay et al., 2015], or PETSc, is employed.

PETSc is entirely implemented in C, although its Python interface *petsc4py* makes it accessible by a framework like Firedrake. It provides a wide range of parallel algorithms for solving linear and nonlinear systems, as well as a considerable number of options to drive the solvers. 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 by and shares many ideas with OP2[1] [Giles et al., 2011], although it differs in a few yet significant ways. In this section, we first describe the common features and then focus on the aspects that are peculiar of PyOP2.

Programming Model

OP2 offers abstractions for modeling an unstructured mesh, in terms of *sets* (e.g. vertices, edges), *maps* between sets (e.g., a map from edges to vertices to express the mesh topology), and *datasets* associating data to each set element (e.g. 3D coordinates to each vertex).

OP2 programs, like the one in Listing 4, are expressed as sequences of parallel loops, or `op_par_loop`. Each loop applies a computational *kernel* to every element in a mesh set. These kernels can access data associated to either the loop iteration set (direct access) or to other sets (indirect access) through maps. Maps are implemented as arrays of indices, so an indirect access requires, from a computational viewpoint, an additional memory access (e.g., `A[B[i]]`).

[1] 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 (Py)OP2 program. OP2 syntax is used.

```
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))
```

The running example includes three parallel loops. Let us focus on the first of them. This loop iterates over the `edges` set (whose declaration is omitted for brevity), as indicated by the first parameter of the `op_par_loop` function. The loop applies `kernel1` to every element in the indicated iteration space, i.e. to each edge. `kernel1` reads data associated to an edge (direct access) and increments the two adjacent vertices with the read value (indirect access). This information, essential for parallelization and optimization, is indicated to OP2 through the *access descriptors*, or `op_arg_dat`. The first access descriptor uses the special keyword OP_ID for the data array `x`, which simply means that `x` is being accessed directly, i.e. with the identity function. In addition, it tells through `OP_READ` that such access is of type read. The second and third access descriptors express that `temp`, a dataset associated with vertices, will be incremented (`OP_INC`) by `kernel1`, and that this increment occurs indirectly through the map `edges2vertices`. The index array `edges2vertices` maps each edge index into the indices of its two incident vertices, with the integer values of 0 and 1 in the access descriptors indicating which of the two vertices should be considered.

Execution Model for Shared-Memory Parallelism

A fundamental property is that the execution order of parallel loop iterations does not influence the result. The shared-memory parallelization of a loop in OP2 (and PyOP2) exploits this property. In essence, the iteration space is partitioned and different partitions are executed concurrently by as many threads, with the restriction that indirect increments to data values shared by two or more partitions must be serialized.

Consider again the first loop of the running example. The `edges` iteration set is partitioned. Partitions that share values associated with `temp` are considered adjacent (from a topological point of view, these are the partitions connected through one or more vertices). The accesses to `temp` are of type increment, so the adjacent partitions are considered conflicting and are assigned a different color. At run time, partitions with the same color can be executed in parallel, while partitions with different colors will be scheduled serially. The execution of elements inside a partition is serialized, since each partition is executed atomically by a single thread. This scheme prevents race conditions by construction.

This process is applied on a per-loop basis and is implemented exploiting the information provided through the access descriptors. Optionally, external libraries can be used to reorder the partitions thus maximizing iteration locality.

Execution Model for Distributed-Memory Parallelism

Distributed-memory parallelism is conceptually simpler than the shared-memory counterpart. During the OP2 initialization phase, sets, maps, and datasets are partitioned and then distributed to different processes. For executing a parallel loop, MPI messages may need be exchanged to update any out-of-date values along the partition boundaries. This phase is overlapped with the execution of a set of local, or "core", iterations. Once both phases have finished, the remaining boundary iterations are computed.

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.

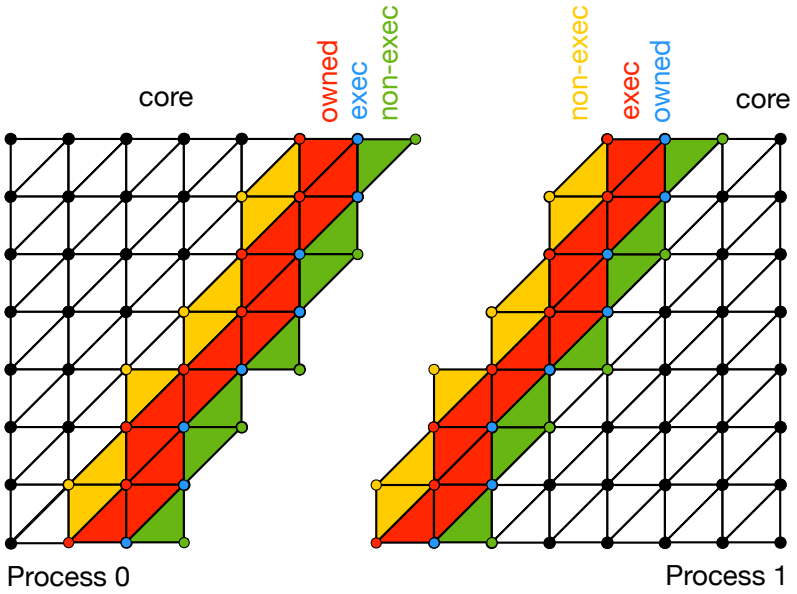


Figure 2.2: Distributed-memory parallelism in OP2 and PyOP2. Matching colors represent identical subsets of iterations. The image was inspired by an example in Rathgeber [2014].

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 2.2. 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 ways.

- An OP2 program is analyzed statically. A source-to-source compiler produces a legal C program, which can be subsequently compiled and executed. PyOP2, on the other hand, is entirely implemented in Python, so code generation occurs at run-time. One problem with OP2 is that its source-to-source compiler is quite limited in the analysis of input programs. PyOP2 does not have this problem, because

objects like sets, maps and parallel loops are constructed and inspected during the interpretation of the source program itself. This makes PyOP2 easily composable with other layers, as proven in the context of the Firedrake project. A hierarchy of "software caches" is however needed in PyOP2 to minimize the overhead due to redundant actions (e.g., generating code each time the same parallel loop is encountered).

- 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 the runtime analysis.
- 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 mesh renumbering for increased data locality. PyOP2, as explained in Section 7, relies on an external software module, DMPlex, for these functionalities. DMPlex is much more versatile than OP2, and this plays a key role when implementing 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 mesh element (e.g., vertex, cell) and its neighboring elements. In the literature, the word "stencil" is often use to refer to the special case in which the rule for accessing the neighboring elements is expressed as an affine function. This is the case of computational methods on structured meshes, like those based on finite difference. However, since this thesis focuses on unstructured meshes, we prefer to generalize this notion. In particular, we characterize a stencil as follows:

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 $f_j(\boldsymbol{i}) = \boldsymbol{i} * h + o$, with $h, o \in \mathbb{N}$.

Stencils for unstructured meshes Given an element $\boldsymbol{i}$ in the mesh and an affine access function g, a stencil is a vector-valued operator $f(\boldsymbol{i}, g) = [f_1(\boldsymbol{i}, g), f_2(\boldsymbol{i}, g), ..., f_n(\boldsymbol{i}, g)]$ 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 $f_j(\boldsymbol{i}, g) = g(\boldsymbol{i}) * h + o$, with $h, o \in \mathbb{N}$.

Stencil Languages for Unstructured Meshes

OP2 is an example of a language for implementing stencil-based computations on unstructured meshes. Maps in the OP2 language are used to express non-affine stencils.

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 on 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[2] 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

[2]The example was extracted from Rathgeber [2014]

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 [Brandvik and Pullan, 2010] is a Python framework for expressing computations using 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 [Unat et al., 2012]. Other stencil languages relying on auto-tuning systems for increasing the performance of the generated code were presented in Zhang and Mueller [2012], Datta et al. [2008], Christen et al. [2011].

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 was presented in Henretty et al. [2013].

In spite of a large research effort, it is however not clear whether and to what extent these 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 compilers and libraries for loop optimization. This will provide the necessary background for the contributions presented in Chapters 3 and 5.

2.3.1 Loop Reordering Transformations

The study of high level transformations for the optimization of loops in imperative languages dates back to the sixties. The initial focus was on techniques for improving data locality in loop nests; this was motivated by

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 Bacon et al. [1994]. Over the last twenty years, there has been a great deal of effort in trying to improve the effectiveness of loop transformations, as well as in developing tools for automating them. Excellent results have been achieved by general-purpose compilers, which can nowadays count on powerful optimization engines (especially the *Intel* and *Cray* compilers). Polyhedral compilers, discussed in Section 2.3.3, have so far achieved controversial results, instead.

A class of optimizations relevant for this thesis is the one based on the *reordering* of loop nest iterations. Among them, it is worth mentioning the following:

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 a loop is "reversed", the order in which its iterations are executed is flipped. For instance, the reverse of a loop iterating from 0 to $n-1$ with unitary increment is a loop from $n-1$ to 0 with unitary decrement. This transformation can enable other transformations or, under certain circumstances, eliminate the need for some temporary variables.

Skewing Loop skewing, often also called "time skewing", aims to improve data reuse and to expose parallelism in wave-front computations. In these computations, 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][j] = f(A[i-1][j], A[i][j-1])`).

Fission Sometimes also referred to as loop distribution, loop fission "splits" a loop into a sequence of loops. The new loops have the same iteration space as the original one, but include only a subset of statements. This may improve or even affect data locality in a loop accessing a significant number of datasets. Increase in loop overhead is another negative effect.

Further, to this class of optimizations belong the two fundamental loop reordering transformations studied in Chapter 3 in the context of unstructured stencils:

Tiling Also known as *blocking*, loop tiling is probably one of the most studied and powerful transformations. Its main goal is improving data locality, although it is also sometimes used to extract parallelism. The basic idea is to "chunk" 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 5. Here, square-shaped tiles are executed with the objective of increasing data reuse along the `j` dimension. Depending on properties of the loop nest such as data dependency pattern and control flow, automating or even just implementing tiling may pose significant challenges. For example, tiling a loop nest in presence of stencils is usually a non-trivial problem. Early work on this trasformation was presented in **??**. More recent studies tackling, among the various aspects, automation and effectiveness of tiling algorithms, have been conducted by **???**. The impact of the tile shape has been investigated by Grosser et al. [2014] and **?**.

Fusion A sequence of loops can be fused, or "merged", to improve data locality and reduce loop overhead. In the simplest instance, all loops in a sequence have the 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 such a case, applying loop fusion is straightforward. In general, however, loops can have different bounds and the data dependence pattern among a set of statements may be non-trivial. In these cases, loop fusion becomes more challenging, or simply not feasible. The loop fusion problem has been tackled formally by Darte [2000].

[3] The example is partly extracted from `www.netlib.org/utk/papers/autoblock/node2.html`

LISTING 5: Illustration of loop tiling in a classic matrix-matrix multiplication kernel[3]. The matrices are square of size $N \times N$. If b is chosen small enough to fit some level of cache, data reuse can be achieved.

```
// Original implementation

for i = 1 to N, 1
  for j = 1 to N, 1
    for k = 1 to N, 1
      C[i][j] += A[i][k]*B[k][j];

// Tiled implementation

for i0 = 1 to N, b
  for j0 = 1 to N, b
    for k0 = 1 to N, b
      for i = i0 to min(i0+b-1, N)
        for j = j0 to min(j0+b-1, N)
          for k = k0 to min(k0+b-1, N)
            C[i][j] += A[i][k]*B[k][j];
```

2.3.2 Composing Loop Tiling and Loop Fusion

Reordering transformations can be composed for improving their effectiveness. A case of particular interest for this thesis is the composition of loop fusion and loop tiling, in which a sequence of loops is fused by grouping multiple blocks of iterations into tiles. Such a transformation can turn data reuse across consecutive loops into data locality, thus optimizing memory bandwidth and memory latency. Composing loop fusion and loop tiling is however complicated; automation is even more challenging, especially depending on the kind of stencils that need be supported.

In the domain of computational methods for approximating the solution of PDEs, the composition of loop fusion and loop tiling is often referred to as *time tiling*. In these codes, a sequence of loops over the spatial dimensions is repeatedly executed within a time loop; time tiling fuses the spatial loops by building tiles crossing the time dimension. Time tiling has extensively been studied for structured stencil codes [Bondhugula et al., 2014, Holewinski et al., 2012, Zhou et al., 2012, Bondhugula et al., 2008]. It has instead received very limited attention in the context of unstructured stencil codes – where it is also known as *sparse tiling* – because of the complexity inherent in performing data dependency analysis. Prior to this thesis, sparse tiling had only been applied "manually" to individual benchmarks, through "ad-hoc" strategies. In Chapter 3 we advance the state-of-the-art by introducing:

- A technique for applying sparse tiling to *arbitrary* sequences of loops that are expressible with the *loop chain abstraction* [Krieger et al., 2013]. This contribution was a joint effort with the authors in Strout et al. [2014].
- A system that automates this technique and enables execution on distributed-memory architectures.

With plain loop tiling (i.e., in absence of loop fusion), a tile can be represented as a single block of iterations. This has been explained in the previous section and illustrated in Listing 5. Time/sparse tiling generalize this idea. The fused loops may either differ in iteration space or be characterized by non-trivial data dependencies. For constructing a legal scheduling, tiles need be assigned multiple blocks of iterations, one for each of the fused loops. To this purpose, there exist two different approaches:

Split tiling Tiles are constructed such that all data dependencies are satisfied and are executed according to a partial ordering. Many split tiling schemes have been studied for structured stencil codes. These schemes, which take the name of the tile shape they end up producing (such as *hexagonal* or *diamond* tiling), differ in the achieved trade-off between parallelism and data locality. The geometrical shape of a tile can be visualized by plotting the fused iteration space and by grouping iterations based on the tile they belong to.

Overlapped tiling Different tiles share subsets of iterations. A shared iteration is "owned" by only one tile and is executed redundantly by the set of overlapping tiles. These tiles store intermediate values into "ghost" regions of memory. This approach relieves from the need for a partial execution ordering, at the price of redundant computation.

Some of these approaches have been reviewed in **?**. In this thesis, we will employ a mixed split-overlapped sparse tiling scheme for unstructured stencil codes, suitable for execution on distributed-memory architectures.

2.3.3 Automation via Static Analysis

There are two alternatives for automating loop reordering transformations. 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) based on a graph-like data structure (e.g., the static single assignment form, or SSA, in LLVM). Some kind of data dependence analysis asserts the legality of a transformation, while cost models are used to predict its impact on performance. Some compilers can expose the cost models directly to the user by textual reports. Users have some form of control over the optimization process: pragma directives can be used to choose explicitly how to optimize a loop nest (e.g., to set a specific unroll factor or to enforce vectorization), while compiler parameters allow tuning global optimization heuristics.

Polyhedral Model Several research compilers, and more recently also a fork of the LLVM compiler through a module called Polly [Grosser et al., 2012], can introduce 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 must be expressions affine in the loop indices and (ii) pointer aliasing must be 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 the polyhedron is available, different scheduling functions can be compared and applied.

If, on one hand, general-purpose compilers using graph-based IRs have now reached an impressive level of sophistication (our experience with the Cray and Intel autovectorization systems is remarkably positive), there is still quite a lot of debate on the effectiveness of polyhedral compilers. One of the main reasons is the lack of evidence: the performance evaluation of state-of-the-art polyhedral compilers is typically conducted on simplistic benchmarks that do not reflect the complexity of real-world applications. We will expand on this matter in Section 3.3.

The fact that we focus on unstructured stencil codes, which render the loop nests non-affine, precludes the adoption of polyhedral compilers. Recently, there has been an effort in extending the polyhedral model to non-affine loops [Venkat et al., 2014], but we hardly believe that this technology will ever be applicable in the real-world applications 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 inspector/executor strategies, as originally proposed by Salz et al. [1991]. Informally, an inspector is an additional piece of code that, at run-time, 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 performs the iterations in a different order exploiting the information produced by the inspector. Examples of reordering transformations through inspector/executor schemes were provided by Strout et al. [2003].

To automate a reordering transformation in presence of unstructured stencils, a compiler replaces loops with suitable inspector/executor schemes. Prior to this thesis, no compilers were capable of introducing complex transformations such as sparse tiling.

2.4 Domain-Specific Optimization

By restricting the attention to relatively narrow classes of programs, it may be possible to identify optimization opportunities that will be missed by general-purpose or polyhedral compilers. This is because, for instance, such optimizations rely on special mathematical properties of the problem being tackled. Most of the contributions in Chapters 3, 4 and 5 exploit this observation. In this section, we review some of the domain-specific optimization systems that inspired our work.

2.4.1 Tensor Contraction Engine

The Tensor Contraction Engine (TCE) is a successful project that turns the mathematical structure of expensive computations in the field of quantum chemistry into powerful optimizations [TCE]. These codes need execute long 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 transformed into low-level code while undergoing several optimization steps. Transformations aimed at reducing the operation count [Hartono et al., 2006, 2009] and finding the best trade-off between redundant computation and data locality [Lam et al., 2011], as well as low-level optimization [Lu et al., 2012] are applied in this framework. Many 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

Ragan-Kelley et al. [2013] recently introduced Halide, a high level language for expressing image processing kernels. The run-time optimization system, which relies on auto-tuning to explore the transformation space, has been demonstrated to achieve a performance at least comparable to that of hand-written (and hand-optimized) codes, and in many cases to outperform them. Halide is a successful contribution to the landscape of domain-specific languages, as it is currently employed for development of production code by several companies[4].

In Halide, an image processing pipeline is a sequence of interconnected stages. Each stage applies a numerical kernel – usually a structured stencil – to its input. Numerical kernels are pure functions applied over a 2D domain representing the image being processed. In realistic cases, an

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

image processing pipeline can be quite complex and include even up to a hundred 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.

2.4.3 Spiral

Spiral is a pioneering project on automated code generation starting from a high level specification of a mathematical problem [Püschel et al., 2005]. 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 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 provides feedback to the system so that increasingly optimized implementations can be generated.

Spiral leverages the mathematical specification to apply powerful optimizations. The rewrite rules system itself is essential to simplify complex formulae. Another example is the loop fusion framework [Franchetti et al., 2005], which exploits mathematical properties to detect loop fusion opportunities that would be missed by lower level compilers.

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 exceed 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 over the last decade. In the domain of scientific computing, it is worth mentioning the technique employed by the finite element code *nek5000* [Paul F. Fischer and Kerkemeier, 2008] 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 of such routines, based on the size of the input matrices. A higher level approach has recently been presented in Spampinato and Püschel [2014], in which 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 [Hennessy and Patterson, 2011]. 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 the aforementioned compilers from all other higher level compilers, such as those based on the polyhe-

dral model or those used for translating domain specific languages.

(Auto)vectorization Vectorization is a well-known paradigm that generalizes computation on scalars to computation on vectors – that is, arrays 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 autovectorization, since SIMD instructions are generated without user intervention. When possible and if demonstrated to be effective, autovectorization should be preferred over explicit vectorization for portability reasons. Autovectorization is typically applied to inner loops, although block vectorization [Larsen and Amarasinghe, 2000] is also supported by more advanced compilers (e.g., Intel's).

Performance and Cost Models

Memory pressure This is often used 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.

Memory- and CPU-boundedness 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, target instead 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* [Williams et al., 2009]. 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.

Miscellanea

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.

Chapter 3

Automated Sparse 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 their working set is usually too big to fit some level of cache. 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 this chapter, it is however extremely challenging to implement these optimizations in 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/executor 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 2.3.2 that sparse tiling aims to exploit data reuse across consecutive loops by composing two transformations, namely loop fusion and loop tiling.

Summarizing, 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 a general technique, applicable to arbitrary computations on unstructured meshes.
- Automation is more than a desired feature because application specialists will avoid complicated optimizations that may harm the comprehensibility of source code. We therefore aim for a fully-automated framework, based upon a mixed compiler/library approach.
- Very few studies have addressed the problem of fusing loops when these need be interleaved by routines for message passing. We are aware of none for the scenario in which the memory accesses pattern is irregular. We fill this gap by introducing a sparse tiling scheme that uses redundant computation over partition boundaries to delay communication. The importance of this contribution stems from the fact that most scientific simulations require distributed-memory parallelism.

3.2 Open Problems, Questions, Hypotheses

Loop fusion and loop tiling have widely been studied in the literature. Over the years, a considerable number of compilers and techniques have been proposed for automating these transformations. Their evaluation, however, has traditionally been limited to a small set of benchmarks (or "mini-applications") and single-node performance. This unfortunately does not shed light on their applicability in real-world applications, where the loop nests are often deeper, less structured, and characterized by irregular control flow. On the other hand, it is true that simple stencil codes arising in finite difference methods [Zumbusch, 2013, Holewinski

et al., 2012, Bondhugula et al., 2014], linear algebra routines [Buttari et al., 2008, 2009], and image processing kernels [Ragan-Kelley et al., 2013] can benefit from transformations like fusion and tiling. Since numerical methods for 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 in real-life numerical methods for solving PDEs, and should we expect any gain in performance?

Lack of evidence Why, despite decades of research, loop transformations are rarely used in production code for scientific simulations?

Challenges What are the theoretical and technical challenges that we have to overcome to automate sparse tiling?

The context in which we tackle these questions is the following:

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 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 terabytes of data, hence execution on multi-node systems is required. Any loop transformation we consider will have to support distributed-memory parallelism.

Automation, but no legacy code Sparse tiling is an "extreme optimization". It requires a great deal of effort to be implemented as it imposes a thoughtful restructuring 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 domain-specific

languages, which abstracts sparse tiling through a simple interface (i.e., a single construct to tell the compiler "transform 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, which makes automation almost impossible.

Based upon these observations and requirements, we decompose our problem into two tasks:

1. writing a library for expressing inspector/executor schemes enabling sparse tiling in arbitrary computations on unstructured meshes;
2. integrating the library with a multilayer framework that relies on domain-specific languages.

Before addressing these two tasks, respectively in Sections 3.5-3.8 and Section 3.9, we elaborate on the theoretical and technical challenges that arise when applying loop fusion and loop tiling to real codes (Section 3.3), and review the related work (Section 3.4).

3.3 Applying Fusion and Tiling is More Difficult than Commonly Thought

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

We identify three classes of problems that are often 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.

Computational boundedness Computational methods for PDEs are structured as sequences of loop nests, each loop nest characterized by its own operational intensity. Within the same application, some loop nests may be memory-bound, while others

LISTING 6: The "bare" structure of a numerical method for solving a PDE. Three parallelizable sweeps over the mesh – in particular, over cells, nodes, and boundary nodes – are executed within a time-stepping loop. In the cells loop, the invocation of a kernel is shown. First, the memory indirections are resolved and data is moved into suitable buffers. The data passed to the kernel is now contiguous in memory, which hopefully maximizes the chances of vectorization. Once the kernel has been executed, the computed values are "scattered" back from a buffer to memory. Distributed-memory parallelism is achieved through MPI, in particular through the `MPI_Comm (...)` calls that separate different mesh sweeps. Additional calculations, for instance in `Calc (...)`, could also interleave the execution of consecutive loops.

```
1  // Time-stepping loop (T = total number of iterations)
2  for t = 1 to T {
3    // 1st sweep over the C cells of the mesh
4    for i in C {
5      buffer_0 = {0.0} // will store the kernel output
6      buffer_1 = gather_data ( A[f(map[i])], ...  )
7      ...
8      kernel_1( buffer_0, buffer_1, ...  );
9      scatter_data ( buffer_0, f(map[i]) )
10   }
11   Calc (...);
12   MPI_Comm (...);
13   // 2nd sweep over the N nodes of the mesh
14   for i in N {
15     ...  // Similar to sweep 1
16   }
17   // Boundary conditions:  sweep over the BV boundary nodes
18   for i in BV {
19     ...  // Similar to sweep 1
20   }
21   ...
22   Calc (...);
23   MPI_Comm (...);
24   ...
25 }
```

CPU-bound. This clearly depends on the numerical method itself, particularly on aspects such as the arithmetic complexity of the operators and the type of discretization employed (e.g., polynomial order of function spaces). Obviously, if most loop nests are CPU-bound, the benefits of sparse tiling on data locality will provide marginal gains. Before even thinking about aggressive optimizations, it is fundamental to determine the bottlenecks of an application. This boils down to answering two questions: (i) what fraction of the execution time is due to memory-bound loop nests; (ii) can CPU-boundedness be re-

lieved by applying other optimizations (e.g., vectorization).

Loop tiling vs space filling curves Loop tiling and space filling curves (SFCs) are two different attempts to solving the same problem: improving the performance of mesh-based computations by increasing data locality. SFCs provide an effective way of iterating over a mesh by guaranteeing that consecutive iterations correspond to neighboring elements; if these elements share some data values, data reuse is exploited. Just like loop tiling, SFCs can be considered loop reordering transformations (see Section 2.3.1). Unfortunately, there seems to be a lack of communication between the communities that have contributed to these two classes of transformations, since we could not find studies comparing their effectiveness. In this chapter, we will integrate SFCs with tiling.

Practical issues Recent work on fusion and tiling for structured mesh applications have addressed automation (e.g., polyhedral compilers), composition of transformations (e.g., time tiling), techniques for minimizing communication (e.g., diamond tiling). However, the following issues were rarely given the attention they actually deserve.

Unstructured meshes Although ad-hoc inspector-executor strategies for some proxy applications had previously been developed, general techniques for arbitrary computations on unstructured meshes have been missing until this thesis[1]. As already explained, the main problem with unstructured meshes is the presence of indirect memory accesses, which complicates the data dependence analysis needed for applying loop transformations.

Time tiling and distributed-memory parallelism We reiterate the fact that real-world computations require large-scale distributed-memory architectures. As Listing 6 shows, MPI calls are now necessary in between consecutive mesh sweeps. This poses a big challenge to time tiling, because now all tiles close to the boundary

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

of a mesh partition require special handling.

Time tiling and extra code The `Comp(...)` function in Listing 6 denotes the possibility that additional computation is performed between consecutive mesh sweeps. `Comp` could represent, for instance, check-pointing, I/O, or the resolution of a linear system through a function call to an external library. Moreover, conditional execution of loops (e.g., through `if-then-else`) may be present. The presence of additional code between loops make loop fusion extremely challenging.

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 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 dependencies in programs structured as arbitrary sequences of loops. We will detail the 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 exploiting data reuse and exposing shared-memory 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 a 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*. In this work, 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 not capable of scheduling across heterogeneous loops (e.g., one over cells and another over degrees of freedom). 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 performs partitioning with the goal of minimizing inter-process communication, 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 does not introduce any sort of loop reordering transformation.

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 in 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 Bondhugula et al. [2014] to a proxy application, but experimentation was limited to shared-memory parallelism.

Overlapped Tiling

In structured codes, multiple layers of halo, or "ghost" elements, are often used to reduce communication [Bassetti et al., 1998]. Overlapped tilling (see Section 2.3.2) exploits the very same idea: trading communication for redundant computation along the boundary [Zhou et al., 2012]. Several works tackle 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 category. To the best of our knowledge, overlapped tiling for unstructured mesh applications has only been studied analytically,

by Giles et al. [2012]. Further, we are not aware of any techniques for automation.

3.5 The Loop Chain Abstraction for Generalized Inspector/Executor Schemes

In this section, we formalize the loop chain abstraction for unstructured mesh applications and discuss its relationship with inspector/executor schemes.

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 reiterate once more that the presence of indirect memory accesses inhibit static loop optimization for data locality. The idea pursued in this chapter is to replace static with dynamic optimization, exploiting the information carried by the loop chain. The loop chain must somehow be added to or derived from the input code. The inspector performs data dependence analysis using the information carried by the loop chain and produces a loop reordering, which implements a sparse tiling schedule. This schedule is used by the executor, a piece of code that replaces the original sequence of loops.

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

- The computation of the inspection phase introduces an overhead. In many scientific computations, the data dependence pattern is static – or, more informally, "the mesh does not change over time". This means that the inspection cost may be amortized over multiple iterations of the executor. If instead the mesh changes over time (e.g., in case of adaptive mesh refinement), a new inspection must be performed.

- To write an inspector/executor scheme for sparse tiling there are several options. One possibility is to provide a library and leave the application specialists in charge of rewriting their code. As observed in the previous sections, this is a very optimistic perspective. An alternative is to raise the level of abstraction: programs can be written in a domain-specific language; loop chain, inspector, and executor can then be automatically derived at the level of the intermediate representation. The tools developed in this thesis enable both approaches.

We will further elaborate on these points in the later sections.

3.5.2 Definition of a Loop Chain

In Krieger et al. [2013], a loop chain is defined as follows:

- A loop chain $\mathbb{L}$ consists of n loops, $L_0, L_1, ..., L_{n-1}$. There are no global synchronization points in between the loops. The execution order of a loop's iterations does not influence the result.

- $\mathbb{D}$ is a set of disjoint m data spaces, $D_0, D_1, ..., D_{m-1}$. Each loop accesses (reads from, writes to) a 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 $W_{L_l \to D_d}(\vec{i})$ are access relations for a loop L_l over a data space $D_d \in D$. They indicate which locations in the data space D_d an iteration $i \in L_l$ reads from and writes to, respectively. A loop chain must provide all access relations for all loops. For example, if L_l writes to the array `A` as `A[B(i)] = f(...)`, then the loop chain will have to provide an access relation $B_{L_l \to A}(\vec{i})$.

3.5.3 The Abstraction Revisited for Unstructured Mesh Applications

Motivated by the issues raised in Section 3.3 and inspired by the programming and execution models of OP2 (reviewed in Section 2.2.2), we revisit the loop chain abstraction. This new definition is more suitable for real-world unstructured mesh applications.

- A loop chain $\mathbb{L}$ consists of n loops, $L_0, L_1, ..., L_{n-1}$. There are no global synchronization points in between the loops. The execution order of a loop's iterations does not influence the result.

- $\mathbb{S}$ is a set of disjoint m sets, $S_0, S_1, ..., S_{m-1}$. Sets are used to define iteration spaces. Possible sets are the cells in the mesh or the degrees of freedom associated with a function.

  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}). As previously shown in Figure 2.2, S^{owned} are iterations that belong to P which will be executed redundantly 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 – a property that enables executing tiles without the need for synchronization.

  S^{ne}: these are iterations of other processes that are communicated to P because they need be 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 to some other sets or the special placeholder $\perp$, which indicates that the access is direct to some data associated with S. `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 crucial difference is the presence of sets in place of data spaces. In unstructured mesh applications, a loop tends to access multiple data spaces associated with the same set. The idea is to rely on sets, rather than data spaces, to perform data dependence analysis. This can significantly improve the inspection cost, because typically $|\mathbb{S}| << |\mathbb{D}|$.

Another important difference is the characterization of sets into the three regions core, boundary and non-exec. This separation is essential for enabling distributed-memory parallelism. The extent of the boundary regions is captured by the *depth* of the loop chain. Informally, the *depth* tells how many extra "strips" of elements are provided by the neighboring processes. This allows some redundant computation along the partition boundary and also limits the depth of the loop chain (i.e., how many loops can be fused). The role of the parameter *depth* will be clear by the end of Section 3.8.

3.6 Loop Chain and Inspection Examples

Using the example in Listing 7 – a plain C implementation of the OP2 program in Listing 4 – we describe the actions performed by a sparse tiling inspector. The inspector takes as input the loop chain illustrated in Listing 8. We show two variants, for shared-memory and distributed-memory parallelism. The value of the variable `mode` at line 18 in Listing 8 determines the variant to be executed.

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 falling in P_i – or, in other words, the edges in partition P_i – are assigned

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

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

LISTING 8: Building the loop chain for inspection.

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

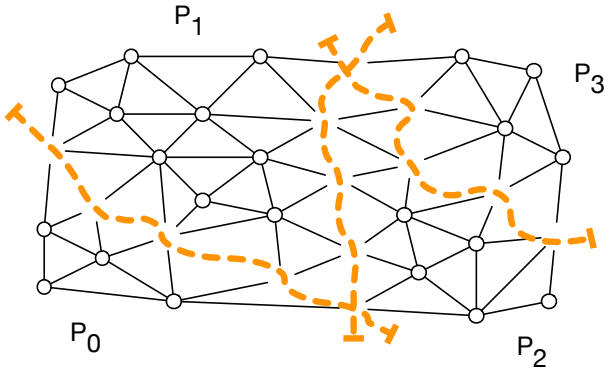


Figure 3.1: Partitioning of the seed loop. The vertices are illustrated to make the connectivity of the mesh clear, although they do not belong to any partition yet.

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 (P_0 and P_3) are not connected through any edge or cell. These four partitions correspond to four tiles, $[T_0,\ T_1,\ T_2,\ T_3]$, with $P_i = T_i$.

As explained in the next two sections, the inspection proceeds by populating T_i with iterations from L_1 and L_2. The challenge of this task is guaranteeing that all data dependencies – read after write, write after read, write after write – are honored. The output of the inspector is eventually passed to the executor. The inspection carries sufficient information for computing sets of tiles in parallel. T_i is always executed by a single thread/process and the execution is atomic; that is, it does not require communication 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.

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 Fig-

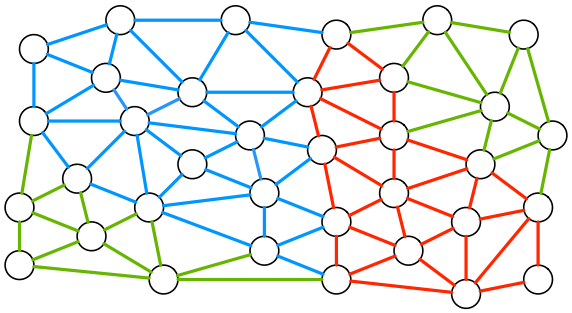


Figure 3.2: A snapshot of the mesh after tiling L_0.

ure 3.2. In the following, with the notation T_i^c we indicate that the i-th tile has color c.

To populate $[T_0^G,\ T_1^B,\ T_2^R,\ T_3^G]$ 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 implies, 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 tiling L_1 and L_2. We assign integer positive numbers to colors to reflect their ordering, where a smaller number means higher execution priority. We can assign, for example, 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 need to compute a *projection* for any write or local reduction performed by L_0. The projection required by L_0 is a function $\phi : V \rightarrow \mathbb{T}$ mapping the vertices in V – as indirectly incremented during the execution of L_0, see Listing 7 – to a tile $T_i^c \in \mathbb{T}$. Consider the vertex v_0 in Figure 3.3. v_0 has 7 incident edges, 2 of which belong to T_0^G, while the 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 $[c_0,\ c_1,\ c_2]$ in Figure 3.3. These three cells have in common the fact that they are adjacent to both green

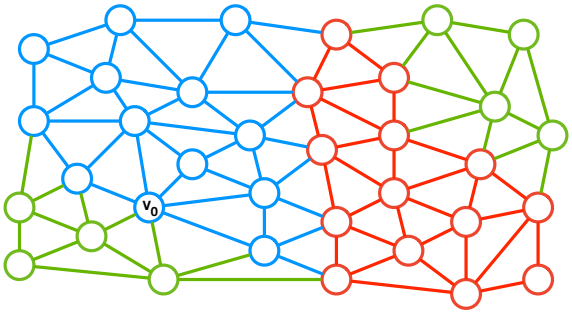


Figure 3.3: Illustration of the projection function computed after tiling L_0. The projection is used by L_1 for a correct scheduling of cells.

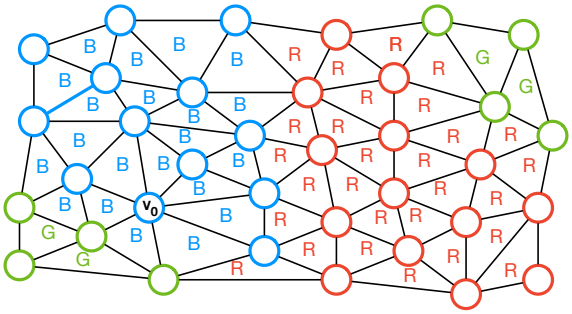


Figure 3.4: A snapshot of the mesh after tiling L_1.

and blue vertices. For c_1, and similarly for the other cells, we compute $\text{MAX}(\phi(v_0),\ \phi(v_1),\ \phi(v_2)) = \text{MAX}(B, G, G) = B = 1$. 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 recall that the execution order, for correctness, must be "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 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 necessary. Figure 3.5 shows the output of this last phase.

Conflicting Colors It is worth noting how T_2^R "consumed" the frontier elements of all other tiles every time a new loop was scheduled. Tiling a loop chain consisting of k loops has the effect of expanding the frontier of a tile of at most k vertices. With this in mind, we re-inspect the loop

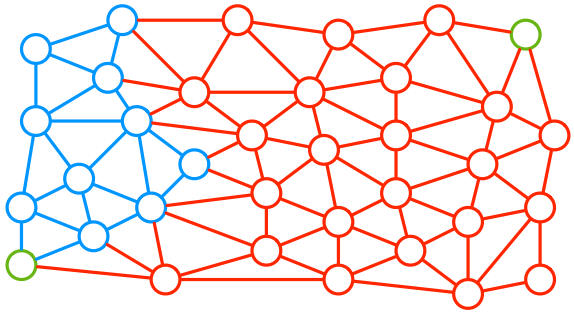
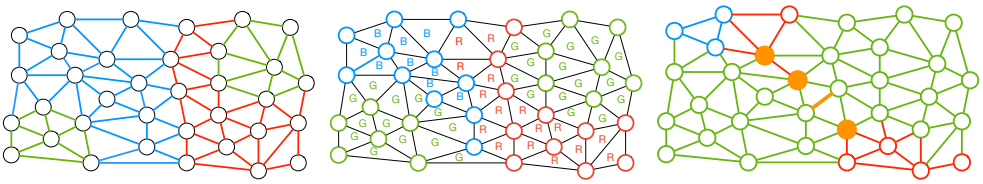


Figure 3.5: A snapshot of the mesh after tiling L_2.

chain of the running example, although this time employing a different execution order – blue (B), red (R), and green (G) – and and a different seed partitioning. Figure 3.6 shows that, by applying the procedure described in this section, T_0^G and T_3^G will become adjacent. This violates the precondition: *tiles can be given the same color, and thus run in parallel, as long as they are not adjacent*. Race conditions during the execution of iterations belonging to L_2 are now possible. We will solve this problem in Section 3.8.1.



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

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

Inspection for Distributed-Memory Parallelism

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

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 been accomplished. With this in mind, we take again L_0 as seed loop and make the following adjustments to the procedure that we have just described.

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: now a single process executes sequentially its own tiles. We can assign colors increasingly, with T_i given color 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 separating the core and boundary regions, we are preventing, by construction, a situation in which a tile crosses the two regions. Further, all tiles within the boundary region are given a color greater than then highest one in core. This constrains the execution order: no boundary tiles will be executed until all core tiles have been computed.

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

From now on, the inspection proceeds as in the case of shared-memory parallelism. The application of the MAX function when scheduling L_1 and L_2 makes higher color tiles (i.e., those having lower priority) "grow over" lower color ones.

Figure 3.7 shows a distribution of the same mesh used in Figure 3.1 to two processes, as well as a possible partitioning into tiles. It is worth observing that: (i) the whole boundary region is expanding over core, which is an essential requirement for correctness; (ii) the boundary region was sized to enable enough redundant computation, which in turn avoids synchronization.

We finally notice that:

Executor On each process, the executor starts with triggering the MPI communications required for executing the boundary tiles. It pro-

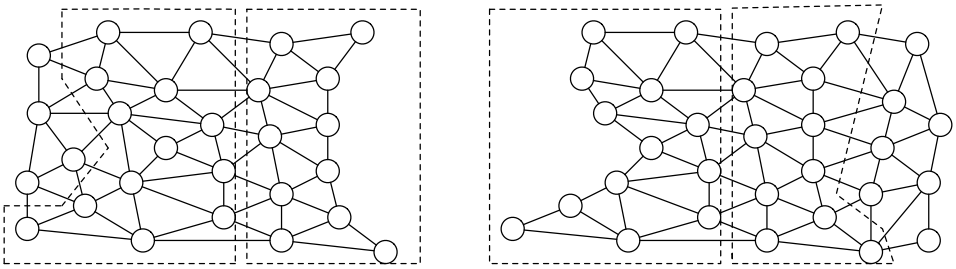


Figure 3.7: Seed partitioning for distributed memory execution.

ceeds to scheduling the core tiles, thus overlapping communication with computation. As all core tiles have finished executing and the MPI communications have terminated, the boundary tiles are computed.

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 **??**, the elements in the green box belong to $\texttt{Process}_1$, but they must also be accessed by $\texttt{Process}_0$ to update all elements in the yellow box correctly. The size of the green box (i.e., the amount of redundant computation) depends on n; that is, on how many loops one wants to fuse.

Efficiency The underlying hypothesis is that the increase in data locality achieved through sparse tiling will outweigh the cost of redundant computation. This is based on the consideration that the core region tends to be way larger than the boundary region. Moreover, not all iterations along the boundary region need be redundantly executed at every loop. The non-exec tile T_{ne} is given the highest color, so it expands over the boundary region as new loops are tiled. This will create some dirty values (until the next round of MPI communications), which however will never be touched since T_{ne} will not be scheduled for execution.

3.7 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:

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

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

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

While the output dependencies between loops L_x and L_y are:

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

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

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

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

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 3.6, 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.

3.8 Formalization

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

Symbol	Meaning
$\mathbb{L}$	The loop chain
L_j	The j-th loop in $\mathbb{L}$
S_j	The iteration space of L_j
S_j^c, S_j^b, S_j^{ne}	The core, boundary, and non-exec regions of S_j
S	A generic set in $\mathbb{L}$
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
ϕ_S	A projection $\phi_S : S \rightarrow \mathbb{T}$
Φ	The set of all available projections
σ_j	A tiling function $\sigma_j : S_j \rightarrow \mathbb{T}$ for L_j
ts	Average tile size

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

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 schemes for partitioning and coloring would be needed to ensure that no iterations in any S_j^b are assigned to a core tile. A limitation of our inspector algorithm in the case of distributed-memory parallelism is that it must be $L_{seed} = L_0$.

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), L_{seed} 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}

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 j = 1 to n − 1 do
10         project(L_{j−1}, σ_{j−1}, Φ, C);
11         σ_j ← tile(L_j, Φ);
12         assign(σ_j, T);
13     end for

14     if has_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 compute_local_maps(T);
20 return T
```

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 are imposing $L_{seed} = L_0$, we here neglect this distinction[2].

[2]The algorithm implemented in the library presented in Section 3.9.1 supports backwards tiling for shared-memory parallelism.

Field	Possible values
region	core, boundary, non-exec
iterations lists	one list of iterations $[T_i]_j$ 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
color	an integer representing the execution priority

Table 3.2: The tile data structure.

Tiles initialization Let ts be the user-specified average tile size. The algorithm starts with partitioning S^c_{seed} into m subsets $\{P_0, P_1, ..., P_{m-1}\}$ such that $|P_i| = ts$ (except possibly for P_{m-1}), $P_i \cap P_j = \emptyset$, and $\cup_{i=0}^{m-1} P_i = S^c_{seed}$. Among all possible legal partitionings, we choose the one that splits S^c_{seed} into blocks of ts contiguous iterations, with $P_0 = \{0, ..., ts-1\}$, $P_1 = \{ts, ..., 2ts-1\}$, and so on. We analogously partition S^b_{seed} into k subsets. We create $m+k+1$ tiles, one for each of these partitions and one extra tile for S^{ne}_{seed}. We therefore have $\mathbb{T} = \{T^c_0, ..., T^c_{m-1}, T^b_m, ..., T^b_{m+k-1}, T^{ne}_{m+k}\}$.

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 given order. This field is set right after the partitioning of L_{seed}, as a tile (by construction) exclusively belongs to S^c_{seed}, S^b_{seed}, or S^{ne}_{seed}.
- The *iterations lists* contain the iterations in $\mathbb{L}$ that T_i will have to execute. There is one *iterations list* $[T_i]_j$ for each $L_j \in \mathbb{L}$. At this stage of the inspection we have $[T_i]_{seed} = [T_i]_0 = P_i$, whereas still $[T_i]_j = \emptyset$ for $j = 1, ..., n$.
- *Local maps* may be used for performance optimization by the executor in place of the global maps provided through the loop chain; this will be discussed in more detail in Section 3.10.2.
- The *color* gives a tile 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, colors are assigned 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 the highest color. The assignment of colors is carried by the function `color` in Listing 1.

Populating tiles by tracking data dependencies To schedule a loop to tiles we use projections. A projection is a function $\phi_S : S \rightarrow \mathbb{T}$. Initially, the projections set Φ is empty. Each time a loop is tiled, Φ may be added some new projections or old projections may be updated. Φ, and consequently the tiling functions for all loops in $\mathbb{L}$, are derived incrementally (within the loop at line 9 in Listing 1) starting from $\sigma_{seed} : S_{seed} \rightarrow \mathbb{T}$, the tiling function of L_{seed}. In the following, we discuss in detail how projections and tiling functions are constructed.

ALGORITHM 2: Projection of a tiled loop

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

```
1  foreach D ∈ L_j.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_t, S_j, values, offset ← inverse_map;
10         φ_{S_t} ←⊥;
11         for e = 0 to S_t.size do
12             for k = offset[e] to offset[e + 1] do
13                 T_last = T[values[k]];
14                 max_color ← MAX(φ_{S_t}[e].color, T_last.color);
15                 if max_color ≠ φ_{S_t}[e].color then
16                     φ_{S_t}[e] ← T_last;
17                 end if
18             end for
19         end for
20         update(C, T, φ_{S_t});
21         Φ = Φ ∪ φ_{S_t};
22     end if
23 end foreach
```

Deriving a projection from a tiling function Algorithm 2 takes as input (the descriptors of) L_j and its tiling function $\sigma_j : S_j \rightarrow \mathbb{T}$ to update Φ. The algorithm also updates the conflicts matrix $C \in \mathbb{N}^{m \times m}$, which indicates whether two tiles having the same color will become adjacent once L_{j+1} is tiled.

A projection tells what tile a set element logically belongs to at a given point of the inspection. A new projection ϕ_S is needed if the elements of S are written by a loop. Let us consider the non-trivial case in which writes or increments occur indirectly through a map $M : S_j \rightarrow S_t^0 \times S_t^1 \times ... \times S_t^{a-1}$. To compute ϕ_{S_t}, we first determine the inverse map (an example is shown in Figure 3.8). Then, we iterate over all elements of S_t and, for each $e \in S_t$, we determine the last tile that writes to e, say T_{last}. This is accomplished by applying the MAX function over the color of the tiles accessing e. We finally simply set $\phi_{S_t}[e] = T_{last}$.

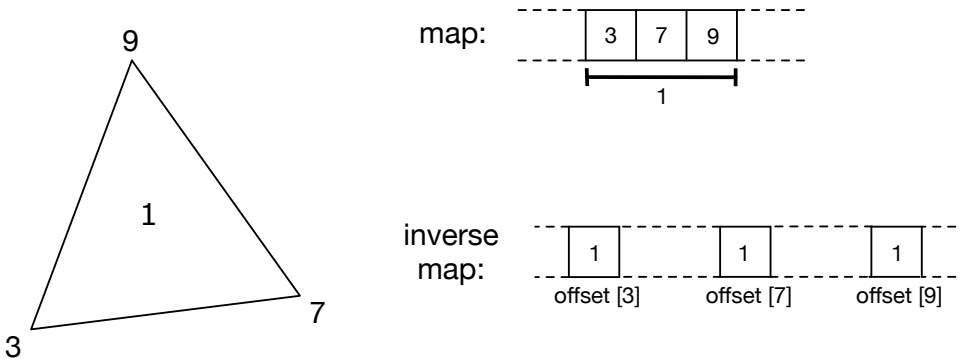


Figure 3.8: Representation of an inverse map. The original map shows that the triangular cell 1 is adjacent to three vertices, namely 3, 7, and 9. The inverse map associates vertices to cells. Since the mesh is unstructured, different vertices can be incident to a different number of cells. The array `offset` determines the distance between two consecutive vertices in the inverse map. For instance, all entries in the inverse map between `offset[3]` and `offset[4]` are cells incident to vertex 3 (in the illustrated case, the first of these is cell 1).

Deriving a tiling function from the available projections Using Φ, we compute σ_j 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_j is inverted and the iterations are added to the corresponding iteration lists $[T_i]_j$, for all $T_i \in \mathbb{T}$.

Detection of conflicts If C indicates the presence of at least one conflict, say between T_{i_1} and T_{i_2}, we add a "fake connection" between these two tiles and loop back to the coloring stage. T_{i_1} and T_{i_2} are now connected, so they will be assigned different colors.

ALGORITHM 3: Building a tiling function

Input: A loop L_j, the projections set Φ
Output: The tiling function σ_j

```
1  σ_j ← ⊥;
2  foreach D ∈ L_j.descriptors do
3      if D.map == ⊥ then
4          σ_j ← Φ[S_j];
5      else
6          arity ← D.map.arity;
7          φ_S ← Φ[D.map.S_j];
8          for e = 0 to S_j.size do
9              σ_j[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_j[e].color, adjacent_tile.color);
13                 if max_color ≠ σ_j[e].color then
14                     σ_j[e] ← adjacent_tile;
15                 end if
16             end for
17         end for
18     end if
19 end foreach
20 return σ_j
```

3.8.2 The Generalized Sparse Tiling Executor

The sparse tiling executor is illustrated in Algorithm 4. It consists of four main phases: (i) boundary exchange between processes through non-blocking MPI communications, (ii) execution of core tiles (in overlap with communication), (iii) waiting for the termination of all MPI communications, (iv) execution of boundary tiles.

As seen in the example in Section 3.6 and based on the explanation in Section 3.8.1, 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 execut-

ALGORITHM 4: The executor algorithm

Input: A set of tiles $\mathbb{T}$
Result: Execute the loop chain

```
1  T^c, T^b ← group_tiles_by_region(T);

2  start_MPI_comm();

3  foreach color do
4  |  foreach T ∈ T^c s.t. T.color == color do
5  |  |  execute_tile(T);
6  |  end foreach
7  end foreach

8  end_MPI_comm();

9  foreach color do
10 |  foreach T ∈ T^b s.t. T.color == color do
11 |  |  execute_tile(T);
12 |  end foreach
13 end foreach
```

ing 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 $n = 1$ (i.e., no loop fusion), only a single "strip" of elements belonging to adjacent processes need be redundantly executed for overlapping computation and communication. As n increases, more off-process data is required. In the example in Figure **??** we had $n = 3$, so three "strips" of off-process elements were necessary for computing correctly the dataset associated with vertices 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 fuse more than *depth* loops, so if n is found to be larger than *depth*, we will need $\lceil \frac{n}{depth} \rceil$ different loop chains.

3.8.3 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.

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 computational costs of the main inspection phases.

3.9 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 2.2.1.

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 **??**.

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

Simplicity of analysis The abstractions used in Firedrake and PyOP2 simplify the analysis on the input programs required for automation.

For example, from the parallel loop construct in PyOP2 (reviewed in Section 2.2.2) we can derive useful information to build a loop chain.

Flexibility As shown in the next sections, it is very simple to use sparse tiling in a Firedrake program. It is sufficient to declare which regions of code should be transformed, and inspector/executor schemes will be derived automatically at the level of the run-time support. In the case of a PyOP2 program, the generation of inspector/executor schemes is only partly automated: the separation of a set into the core, boundary and non-exec regions for distributed-memory parallelism is now user's responsibility. An inspector-executor scheme can also be written from scratch, for instance in a plain C program. In this case, the loop chain must be provided explicitly through direct calls to the SLOPE library.

User base We want generalized sparse tiling to be usable in real software. We therefore chose a framework, Firedrake, with a user base in rapid expansion. This, obviously, increased the implementation effort.

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

3.9.1 SLOPE: a Library for Sparse Tiling Irregular Computations

SLOPE is an open source software that provides an interface to build loop chains and to express inspector/executor schemes for sparse tiling[3].

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 different sets are accessed. The example in Listing 8 illustrates the interface exposed by SLOPE.

SLOPE implements the algorithms in Section 3.8.1. Further, it provides additional features to estimate the effectiveness and to verify the correctness of sparse tiling:

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

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 [Ayachit, 2015].

Inspection summary 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.

3.9.2 PyOP2: Lazy Evaluation and Interfaces

PyOP2 has been described in Section 2.2.2. This section focuses on three complementary aspects: (i) the interface offered for identifying loop chains; (ii) the lazy evaluation mechanism that allows building loop chains; (iii) the interface with SLOPE to execute inspector/executor schemes.

To apply sparse tiling to a sequence of loops, PyOP2 provides the *loop_chain* interface, exemplified in Listing 5. This interface is exposed to both users and Firedrake. In a Firedrake program, parallel loops may actually not appear at all in the source code because they will automatically be generated from the mathematical specification.

LISTING 5: Illustration of the *loop_chain* interface in PyOP2. Apart from the `name` and the `tile_size`, all other parameters are optional, and only useful for performance evaluation (e.g., logging) and tuning. Among the options for performance tuning, there is `split_mode`: this allows the user to specify how an excessively long loop chain should be split into smaller loop chains, for instance imposing a maximum number of loops.

```
1 with loop_chain (name, tile_size, split_mode, ...):
2     Any Python code here, including UFL of PyOP2 loops
```

PyOP2 exploits lazy evaluation of parallel loops to generate an inspector/executor scheme. The parallel loops encountered during interpretation of the input program – or, analogously, those generated through Firedrake – are not executed immediately; rather, they are first pushed into a queue. The sequence of parallel loops in the queue is called *trace*. If a dataset f needs be read, for example because a user wants to inspect its values or a global linear algebra operation needs 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 includes all parallel loops that must be executed for computing f. At this point, the sub-trace can be executed or further pre-processed.

The rule that we employ is that all loops in a trace that were created within a *loop_chain* scope are candidates for sparse tiling. In detail, the interaction between PyOP2 and SLOPE is as follows:

1. The examples in Listing 5 shows that a *loop_chain* defines a new scope. As this scope is entered, a stamp s_1 of the trace is generated. This happens "behind the scenes", because the *loop_chain* is a Python context manager, which can execute pre-specified routines prior and after the execution of the body. As the *loop_chain*'s scope is exited, a new stamp s_2 of the trace is computed. 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: (i) "simple" fusion – all consecutive parallel loops iterating over the same iteration space that do not present data dependencies through indirections are merged into a single loop; (ii) 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. C code implementing an inspector for the loops in the *loop_chain* is returned by SLOPE. PyOP2 compiles this code and executes it; this produces an *inspection* for the loop chain.

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

5. The executor is built in an analogous way to the inspector. It is executed, taking as input an *inspection*, each time a *loop_chain* is exited .

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

Firedrake uses DMPlex [Lange et al., 2015] to handle meshes. In particular, DMPlex is responsible for partitioning, distributing over multiple processes, and locally reordering a mesh. The MPI parallelization, therefore, is managed through Firedrake/DMPlex.

During the start-up phase, each MPI process receives a contiguous partition of the original mesh from DMPlex. An MPI process creates the required PyOP2 sets, which represent either topological 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 sets these regions based on the information provided by DMPlex, and creates the data structures that will enable communication.

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* (the name derives from **?**), 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. In absence of sparse tiling, this value enables overlap of computation with communication when executing a loop, at the price of a small amount of redundant computation along partition boundaries.

3.10 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.

[4]The implementation received invaluable contributions from Dr. Michael Lange.

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.

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.

3.10.1 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 $A\vec{u} = \vec{f}$, 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 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

[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

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).

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 3.10.1.

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 paralleliza-

tion 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 3.8.1 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 3.8.1. 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 3.9 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 3.8.1), for supporting MPI-based executors.

3.10.2 Seigen: an Elastic Wave Equation Solver for Seismological Problems

...

3.11 Conclusions and Future Work

Chapter 4

Minimizing Operations in 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 is implemented in COFFEE and currently in use in the Firedrake framework.

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. We recall from Section 2.2.1 that, 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 specification 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 4.9). Research compilers, for instance those based on polyhedral analysis of loop nests, such as PLUTO [Bondhugula et al., 2008], 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. Olgaard and Wells [2010] show how automated code generation can be leveraged to introduce optimizations that a user should not be expected to write "by hand". Kirby and Logg [2006] and Russell and Kelly [2013] employ mathematical reformulations of finite element integration with the aim of minimizing the operation count. In Luporini et al. [2014] (and in Chapter 5, from which the article is extracted), 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 form compilers, UFLACS [Alnæs, 2016] and TSFC [Homolya and Mitchell, 2016], which particularly target the compilation time challenges of the more complex variational forms. The performance evaluation in Section 4.9 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 executed? 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).

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

In Section 4.2 we introduce a set of definitions mapping mathematical properties to the level of loop nests. This step is an essential precursor to the definition of the two algorithms – sharing elimination (Section 4.3) and pre-evaluation (Section 4.4) – through which we construct the space of legal transformations. The main transformation algorithm in Section 4.6 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 4.7. The numerical experiments are showed in Section 4.9. We conclude discussing the limitations of the algorithms presented and future work.

4.2 Loop Nests, Expressions and Optimality

In this section, we characterize global and local optimality for finite element integration. In order to make the chapter 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 the next sections could also be generalized to larger classes of loop nests, in which basic block independence does not hold, although this would require refinements beyond the scope of this chapter.

We introduce some new concepts by mapping mathematical properties to the loop nest level. We start with the notion of a *linear symbol*. This will

allow us to define a *linear loop* and, more generally, a (perfect) *multilinear loop nest*.

Definition 3 (Linear symbol)**.** A symbol a in an expression e is linear if

1. a is an n-dimensional array, and
2. all occurrences of a in e are indexed through the same vector-valued access function $f = [f_0, ..., f_{n-1}]$ (e.g., $a[f_0(...)][f_1(...)]$), and
3. all sub-expressions of e in which a appears are affine in a.

We discuss some simple examples (to relieve the notation, we set $a[i] \equiv a_i$). The symbol a is *not* linear in any of $a_i b + a_k$, $a_i b + a_{i+2}$, and $a_{ji} b + a_{ki}$, as condition 2 is violated. Condition 3 is violated for a in both b/a_i and $a_i b + a_i a_i$. a is instead linear in $a_i b + a_i$.

Definition 4 (Linear loop)**.** Let L_i be a loop iterating over the space I through the iteration variable i. L_i is linear if in its body

1. i only appears as an array index, and
2. all symbols in which i appears are linear symbols.

Definition 5 (Multilinear loop nest)**.** A multilinear loop nest of arity n is a perfect nest composed of n linear loops.

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 now define two different classes of loops.

Definition 6 (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 7 (Order-free loop)**.** A loop is said to be an order-free loop if its iterations can be executed in any arbitrary order.

```
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 += sum_{w=1}^{m} α^w_eij β^w_eik σ^w_ei
```

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

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 notation into code. In particular, m is the number of monomials (a form is a sum of monomials), α_{eij} and β_{eik} represents the product of a coefficient function (e.g., the inverse Jacobian matrix for the change of coordinates) with some linear symbols (e.g., 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 8 (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) reduction loop, and a multilinear loop nest whose body is an independent basic block in which expressions are in normal form. Test and trial functions (or derivatives thereof) are the linear symbols of the multilinear loop nest.

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

Definition 9 (Optimality of a loop nest)**.** Let Λ be a 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 10 (Flop-decreasing transformation)**.** A transformation which reduces the operation count is called flop-decreasing.

Definition 11 (Local optimality of a loop nest)**.** Given Λ, Λ' and Γ as in Definition 9, 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 9 and 11 do not take into account memory requirements. If the execution of a 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 4.3 – 4.5)
- an algorithm to explore and select points in the transformation space (Section 4.6)

4.3 Transformation Space: Sharing Elimination

We start with introducing the fundamental notion of sharing.

Definition 12 (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 identical sub-expressions, possibly just two symbols.

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 [Luporini et al., 2014], 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

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

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.

4.3.1 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 spatial and temporal sharing through generalized code motion. Spatial sharing is eliminated by capturing common sub-expressions; temporal sharing is eliminated by hoisting expressions out of some loops.

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

In the current example, 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. A second example showing the effect of Strategy 2 was provided in Figure 4.2. In general, choosing between the

two strategies is challenging because multiple factors must be taken into account: the loop sizes, the increase in operations due to expansion, the gain due to code motion, and the presence of common sub-expressions. Before addressing this problem (in Section 4.3.2), we need to understand under what conditions Strategy 2 is applicable. For this, we introduce a relevant class of expressions.

Definition 13 (Structured expression)**.** We say that an expression is "structured in a loop nest Λ" if and only if, for every symbol s_Λ depending on at least one loop in Λ, the spatial sharing induced by s_Λ may be eliminated by factorizing all occurrences of s_Λ in the expression.

Proposition 1. *An expression is structured in a loop nest Λ if Λ is multilinear.*

Proof. This follows directly from Definition 3 and Definition 5, which restrict the number of occurrences of 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, it is guaranteed by Proposition 1 that any sub-expression including at least two instances of the same linear symbol can be applied Strategy 2. As discussed in the next sections, this property will allow us to construct the space of flop-decreasing transformations by "composition" of Strategy 1 and Strategy 2.

Finally, we observe that the σ_{ei} sub-expressions can sometimes be considered "weakly structured". This happens when a relaxed version of Definition 13 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 4.9). Weak structure will be exploited by Algorithm 1 in the attempt to achieve optimality.

4.3.2 Global Analysis of the Expression

To evaluate the global impact of a transformation, we perform a so called *strategy selection analysis* on a *temporaries graph*, which models the input

expression. As we shall see, this outperforms a simplistic approach in which sub-expressions are analyzed in "isolation", because common sub-expressions are taken into account.

It is useful to start with introducing the notion of a *minimal common sub-expression*.

Definition 14 (Minimal common sub-expression)**.** A sub-expression m within an expression e is said to be a minimal common sub-expression (MCS) if:

- m occurs at least twice in e, and
- any m', a sub-expression obtained by removing an operation from m, has no common sub-expressions in e.

For example, in $ab + abc + abcd$ the sub-expression ab is an MCS, whereas abc is not, despite occurring twice.

A temporaries graph is a directed k-partite graph in which vertices are MCSs and edges represent read-after-write dependencies between MCSs (e.g., an edge from e_1 to e_2 indicates that e_1 is read in e_2). The k independent sets $\{T_i\}_{i=0}^{k-1}$ of vertices, or simply *level*s, are determined iteratively. Initially, Strategy 1 is applied to the input expression. This produces T_0 and modifies the input expression by replacing the MCSs in T_0 with appropriate temporaries. Then, Strategy 1 is applied to the modified expression; T_1 and the edges from T_0 to T_1 are in this way determined. This process is repeated k times, until no more MCSs can be extracted. An example of temporaries graph is provided in Figure **??**. It is worth observing that k can be quite large as the input is in normal form. This implies, for instance, that the operands of an inner product occurring multiple times in the variational form will be common sub-expressions in the translated scalar expression.

The *strategy selection analysis* uses a temporaries graph to determine whether applying Strategy 2 to some subsets of levels would lower the global operation count. If the MCSs in the graph are structured expressions, then this analysis can be accomplished in polynomial time (thanks to Proposition 1). Given two levels T_{i_1} and T_{i_2} such that $i_1 < i_2$, one can

1. Starting with $i = i_1$, calculate the total operation count for computing all sub-expressions in $\{T_j\}_{j=i}^{i_2}$. The operation count includes the

loop sizes.

2. Inline the MCSs in T_i into $\{T_j\}_{j=i+1}^{i_2}$. The temporaries in T_i are thus discarded.

3. Apply Strategy 2 to all sub-expressions in $\{T_j\}_{j=i+1}^{i_2}$, factorizing any linear symbol/temporary.

4. Evaluate and store the new total operation count, which includes the loop sizes.

5. Increment i and repeat this process until $i = i_2$.

It is finally straightforward to determine for which range of levels (if any) the application of Strategy 2 improved the operation count. The cost and the effectiveness of the *strategy selection analysis* obviously depend on the number n of pairs $<i_1, i_2>$ that are evaluated (at most $k-1$). Since during our experiments setting $n = k-1$ never led to unacceptable code generation times[3], in the following we assume that the *strategy selection analysis* always considers all legal pairs $<i_1, i_2>$.

4.3.3 The Sharing Elimination Algorithm

Algorithm 1 describes sharing elimination gathering the considerations from the previous sections. It also introduces a model to minimize the operation count within the innermost loop once *strategy selection analysis* has been performed.

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 all *MCS*s. Apply Strategy 1 to these MCSs. This requires introducing some new temporaries. Repeat this process until there are no more *MCS*s.

2. Group the temporaries based on the linear loop they depend on. We have two disjoint sets in the case of bilinear forms and, trivially, a

[3] Even in the case of complex hyperelastic models, the code generation time never exceed a few seconds on a relatively old Sandy Bridge architecture.

single set for linear forms. For each of these sets, build a *temporaries graph* and apply the *strategy selection analysis* discussed earlier.

3. Build the *sharing graph* $G = (S, E)$ for each expression in the body of the multilinear loop nest (these are the expressions transformed by the *strategy selection analysis*, which now include some temporaries). Each $s \in S$ represents a linear 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: this and the following steps will only impact bilinear forms, since otherwise, due to linearity, $E = \emptyset$.

4. Partition S into n disjoint sets $\{S_1, ..., S_n\}$, 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 their factorization 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 linear symbols/temporaries that will be factorized. 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 expression independent of the multilinear loop nest, apply the most profitable between Strategy 1 and Strategy 2.
   Note: this pass speculatively assumes that expressions are (weakly) structured in 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 (step (6)) also attempt to optimize the loop nest globally. We will further elaborate this aspect in Section 4.7.

4.3.4 Examples

In this section, we present a series of examples of increasing complexity.

Example 1

Consider again Figure 4.2a. There are no common sub-expressions, so the *strategy selection analysis* has no effect. The *sharing graph* is $G = (\{b_j, c_i, d_i\}, \{(b_j, c_i), (b_j, d_i)\}$. The ILP formulation leads to the code in Figure 4.2c.

Example 2

In Figure 4.3, Algorithm 1 is executed in a simple yet realistic scenario, which originates from the bilinear form of a Poisson equation in two dimensions. The temporaries graph, which consists of a single level, is used for performing *strategy selection analysis*. This leads to the synthesis in Figure 4.3b. The *sharing graph* is $G = (\{t_0, t_1, t_2, t_3\}, \{(t_0, t_2), (t_1, t_3)\})$, but since there are no factorization opportunities the ILP formulation has no effect.

Example 3

In this example, we focus on the ILP formulation. We consider a bilinear form extracted from a model of temperature-dependent multiphase flow through porous media [Olgaard and Wells, 2010], which we informally refer to as "pressure equation". Although the complete specification of the form is irrelevant for the purpose of this example, it is useful to know that (i) the problem is linear, (ii) there are in total six monomials, (iii) the

```
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++)
    t_det = W_i * det
    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) * t_det
      t3_j = (z1 c_ij + z3 d_ij) * t_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 $I, J, K = 1$.

gradient and the divergence of test and trial functions appear in some monomials (so several distinct linear symbols are present in the input expression).

Figure 4.4a shows the finite element integration loop nest at the end of step (2) in Algorithm 1; that is, once *strategy selection analysis* has been performed. The linear temporaries induce spatial sharing in the loop nest. It is not obvious to individuate the factorization that would result in a local optimum. One can derive the *sharing graph* directly from inspection of Figure 4.4a, obtaining $G = (\{t10_{ik}, t10_{ij}, t11_j, t12_j, t13_k, t14_k, t15_k\}, \{(t10_{ik}, t10_{ij}), (t10_{ik}, t11_j), (t10_{ik}, t12_j), (t15_k, t11_j), (t15_k, t12_j), (t10_{ij}, t14_k), (t10_{ij}, t13_k), (t11_j, t14_k), (t11_j, t13_k), (t12_j, t14_k), (t12_j, t13_k)\})$. The ILP formulation, eventually, retrieves three factorization candidates, as shown in Figure 4.4b.

Example 4

In the last example, we show how the *strategy selection analysis* impacts the operation count in a hyperelastic model. This is the same problem used for performance evaluation in Section 4.9. The code in Figure 4.5a is obtained by repeatedly applying Strategy 1 to the expression in normal form (step (1) in Algorithm 1). The temporaries graph, which consists of 6 levels, can easily be derived. For given values of I and K, the *strategy*

```
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_jk += (((t10_ik * ((t10_ij * t3) + (t4 * ((t11_j * t2) + (t12_j * t1)) * t6))) +
                 (t8 * t9 * t15_k * ((t11_j * t2) + (t12_j * t1)) * t0) +
                 (t10_ij * -1.0 * ((t14_k * t2) + (t13_k * t1)) * c4) +
                 (t7 * ((t14_k * t2) + (t13_k * t1)) * ((t11_j * t2) + (t12_j * t1)) * c4) +
                 (t5 * ((t14_k * t11_j) + (t13_k * t12_j)) * t0))) *
                 W_i * det
```

(a) Before the ILP-driven factorization

```
for (e = 0; e < E; e++)
  ...
  for (i = 0; i < I; i++)
    t_det = W_i * det
    t15 = t1 * t4 * t6
    t16 = (t0 * t5) + (t1 * t1 * t7 * c4)
    t17 = t2 * t0 * t8
    t18 = t2 * t4 * t6
    t19 = t1 * t0 * t8
    t20 = (t0 * t5) + (t2 * t2 * t7 * c4)

    for (int k = 0; k < K; k++)
      t16_k = (t10_ik * t3) + (t13_k * -1.0 * t1 * c4) + (t14_k * -1.0 * t2 * c4)
      t17_k = (t10_ik * t15) + (t13_k * t16) + (t14_k * t2 * t1 * c5) + (t15_k * t9 * t19)
      t18_k = (t10_ik * t18) + (t13_k * t1 * t2 * c5) + (t14_k * t20) + (t15_k * t9 * t17)

    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        A_jk += (t10_ij * t16_k) + (t12_j * t17_k) + (t11_j * t18_k)
```

(b) After the ILP-driven factorization

Figure 4.4: Using the ILP model to factorize the expression arising from the pressure equation.

selection analysis determines that applying Strategy 2 to the temporaries between levels T_0 and T_4 will improve the operation count. Strategy 1 is instead preferable for T_5, which comprises the four temporaries t_{59_k}, t_{60_k}, t_{65_k}, t_{66_k}. The resulting synthesis is shown in Figure 4.5b. Intuitively, instead of executing N operations IK times (Figure 4.5a), now M operations are executed I times (Figure 4.5b). It is true that $M > N$, but the analysis guarantees that I and K are such that the operation count has improved.

In this example, the application of Strategy 2 reduces the operation count by about 1.3×, on average (the actual value clearly depends on I and K). It is not rare, however, to encounter cases (e.g., more complex

hyperelastic models) in which the reduction is even larger than 2×. We emphasize that here we are discussing the improvement from applying Strategy 2 "on top of" Strategy 1; the overall gain due to performing *strategy selection analysis*, which must take Strategy 1 into account, is obviously larger.

```
for (e = 0; e < E; e++)
  ...
  for (i = 0; i < I; i++)
    ...
    for (int k = 0; k < K; k++)
      t40  = ((t8 * t39_k) + (t11 * t38_k))
      t41  = ((t14 * t35_k) + (t15 * t34_k))
      t42  = (t41 * t13)
      t43  = ((t8 * t35_k) + (t11 * t34_k))
      t44  = (t43 * t16)
      t45  = ((t14 * t39_k) + (t15 * t38_k))
      t46  = (t45 * t20)
      t47  = (t40 * t21)
      t48  = (t0 * ((t42 + t44 + t46 + t47)/2))
      t49  = ((t48 + t48) * w1[0])
      t50  = (t43 * t13)
      t51  = (t40 * t20)
      t52  = ((t50 + t50 + t51 + t51)/2)
      t53  = (t41 * t16)
      t54  = (t45 * t21)
      t55  = ((t53 + t53 + t54 + t54)/2)
      t56  = (t24 * (t52 + t55) * t28)
      t57  = (t0 * t55)
      t58  = (t56 + ((t57 + t57) * w1[0]))
      t59_k = (t40 * t37) + (t49 * t20)+
              (t45 * t33) + (t58 * t21)
      t60_k = (t43 * t37) + (t49 * t13)+
              (t41 * t33) + (t58 * t16)
      t61  = (t0 * t52)
      t62  = (t56 + ((t61 + t61) * w1[0]))
      t63  = (t0 * ((t44 + t42 + t47 + t46)/2))
      t64  = ((t63 + t63) * w1[0])
      t65_k = (t40 * t31) + (t62 * t20)+
              (t45 * t23) + (t64 * t21)
      t66_k = (t43 * t31) + (t62 * t13)+
              (t41 * t23) + (t64 * t16)

    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        ...
```

(a) Normal form

```
for (e = 0; e < E; e++)
  ...
  for (i = 0; i < I; i++)
    ...
    c0  = (0.5 * t0)
    c1  = (c0 * ((t14 * t13) + (t8 * t16)))
    c2  = (c0 * ((t14 * t20) + (t8 * t21)))
    c3  = (c0 * ((t15 * t13) + (t11 * t16)))
    c4  = (c0 * ((t15 * t20) + (t11 * t21)))
    c5  = (c0 * ((t11 * t16) + (t15 * t13)))
    c6  = (c0 * ((t8 * t16) + (t14 * t13)))
    c7  = (c0 * ((t11 * t21) + (t15 * t20)))
    c8  = (c0 * ((t8 * t21) + (t14 * t20)))
    c9  = (w1[0] * 2)
    c10 = (t24 * t28)
    c11 = (c10 * ((t20 * t11) + (t21 * t15)))
    c12 = (c10 * ((t13 * t11) + (t16 * t15)))
    c13 = (c10 * ((t20 * t8) + (t21 * t14)))
    ...
    c37 = ((t11 * t37) + (c3 * c36) + (t15 * t33) + (c24 * t16))
    c38 = ((t8 * t37) + (c1 * c36) + (t14 * t33) + (c23 * t16))
    c39 = ((c2 * c36) + (c25 * t16))
    c40 = ((c4 * c36) + (c26 * t16))
    c41 = (c9 * t21)
    c42 = ((c29 * t20) + (c6 * c41))
    c43 = ((c27 * t20) + (c5 * c41))
    c44 = ((t8 * t31) + (c30 * t20) + (t14 * t23) + (c8 * c41))
    c45 = ((t11 * t31) + (c28 * t20) + (t15 * t23) + (c7 * c41))
    c46 = (c9 * t16)
    c47 = ((t11 * t31) + (c27 * t13) + (t15 * t23) + (c5 * c46))
    c48 = ((c28 * t13) + (c7 * c46))
    c49 = ((t8 * t31) + (c29 * t13) + (t14 * t23) + (c6 * c46))
    c50 = ((c30 * t13) + (c8 * c46))

    for (int k = 0; k < K; k++)
      t59_k = (t39_k * c32) + (t38_k * c33) + (t35_k * c35) + (t34_k * c34)
      t60_k = (t35_k * c38) + (t34_k * c37) + (t39_k * c39) + (t38_k * c40)
      t65_k = (t39_k * c44) + (t38_k * c45) + (t35_k * c42) + (t34_k * c43)
      t66_k = (t35_k * c49) + (t34_k * c47) + (t39_k * c50) + (t38_k * c48)

    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        ...
```

(b) After sharing elimination

Figure 4.5: Applying the *strategy selection analysis* to the bilinear form arising from a hyperelastic model in 2D.

4.4 Transformation Space: 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 4.6 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 4.6: Exposing (through factorization) and pre-evaluating a reduction.

Pre-evaluation can be seen as the generalization of tensor contraction (reviewed in Section 2.1.5) 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[4], as a consequence of the results in Kirby and Logg [2007]. 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 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 4.6.

[4]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 Luporini et al. [2016].

4.5 Transformation Space: 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.

4.6 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.

4.6.1 The Main Transformation Algorithm

We address the two following issues:

1. *Coordination of pre-evaluation and sharing elimination.* Recall from Section 4.4 that pre-evaluation could either increase or decrease the op-

eration 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 $\theta : M \rightarrow \mathbb{Z}$ 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 $\theta(m) = \theta^{pre}(m) - \theta^{se}(m)$, where θ^{se} and θ^{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 θ^{se} and θ^{pre} in Section 4.6.2. 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 $D[b] = \theta^{se}(S \cup b_S) + \theta^{pre}(b_P)$.
   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 θ^{se} and θ^{pre} is negligible. We elaborate on this aspect in Section 4.6.2.

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 4.7, 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 4.7: The loop nest produced by the algorithm for an input as in Figure 4.1.

4.6.2 The Cost Function θ

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

We recall that $\theta(m) = \theta^{pre}(m) - \theta^{se}(m)$, with θ^{se} and θ^{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 θ^{se} and θ^{pre} would consist of applying the actual transformations and simply count the number of operations. This would be tolerable for θ^{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 θ^{pre}.

The first step consists of estimating the *increase factor*, ι. This number captures the increase in arithmetic complexity due to the transformations exposing pre-evaluation opportunities. For context, consider the example in Figure 4.8. 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 4.8: 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 $\theta^{pre} = \rho \cdot \iota$. The mass matrix monomial in Figure 4.8 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.

4.7 Formalization

We demonstrate that the orchestration of sharing elimination and pre-evaluation performed by the transformation algorithm guarantees local optimality (Definition 11). The proof re-uses concepts and explanations provided throughout the chapter, as well as the terminology introduced in Section 4.3.3.

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 11 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 4.7). We need to show that there is no other synthesis Λ''_I satisfying Constraint 2 such that $c_{\Lambda''_I} < c_{\Lambda'_I}$ and

that Λ' is achieved through a sequence of flop-decreasing transformations. 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 (at least) a 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 linear 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 in order to determine an optimum which satisfies Constraint 2[5]. Recall that since the number of monomials is in practice very small, this pass can rapidly be accomplished.
2. Consider Algorithm 1. Strategy 1 cannot increase the operation count as both code motion and common sub-expressions elimination are flop-decreasing transformations. Strategy 2, instead, may increase the operation count due to product expansion. If Strategy 2

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

were carelessly applied, for instance as a step of a larger transformation, it would be possible to end up with an increased operational cost. This is avoided by construction (steps (2) and (4)). The *strategy selection analysis*, in particular, compares the operation counts of Strategy 2 and Strategy 1 (a flop-decreasing transformation) and the former is retained if and only if the new state shows an improved operation count. As all transformations that are applied to the input expressions in normal form are of flop-decreasing nature, local optima cannot be prune from the search space.

The ILP model is derived from the *sharing graph* of the transformed expressions. This is used to drive Strategy 2 in the minimization of the operation count within the innermost loop (see Definition 11). At this point, proving the optimality of the innermost loop 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 chapter we have reiterated the claim that Algorithm 3 achieves a globally optimal flop count if stronger preconditions on the 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 4.3.1). This avoids ambi-

guity 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.

4.8 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 chapter. A complete description of the compiler is provided in Chapter 6.

4.8.1 Expressing Transformations with COFFEE

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.

4.8.2 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 Kirby and Logg [2006] modified to produce ASTs rather than strings, to a newly written compiler[6], 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.

4.8.3 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 Olgaard and Wells [2010], 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 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.

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

4.9 Performance Evaluation

4.9.1 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 [Mitchell et al., 2016], PETSc [Smith et al., 2016], petsc4py [Firedrake, 2016], FIAT [Rognes et al., 2016], UFL [Alnæs et al., 2016], FFC [Logg et al., 2016], PyOP2 [Rathgeber et al., 2016b] and COFFEE [Luporini et al., 2016]. The experiments can be reproduced using a publicly available benchmark suite [Rathgeber et al., 2016a].

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[7].

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 Olgaard and Wells [2010], implemented in in the FEniCS Form Compiler.

tens Tensor contraction mode. Work presented in Kirby and Logg [2006], implemented in the FEniCS Form Compiler.

auto Automatic choice between `tens` and `quad` driven by heuristic (de-

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

tailed 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 chapter, 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 4.5, 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 Olgaard and Wells [2010], we vary the following parameters:

- the polynomial degree of test, trial, and coefficient (or "pre-multiplying") functions, $q \in \{1, 2, 3, 4\}$
- the number of coefficient functions nf $\in \{0, 1, 2, 3\}$

While constants of our study are

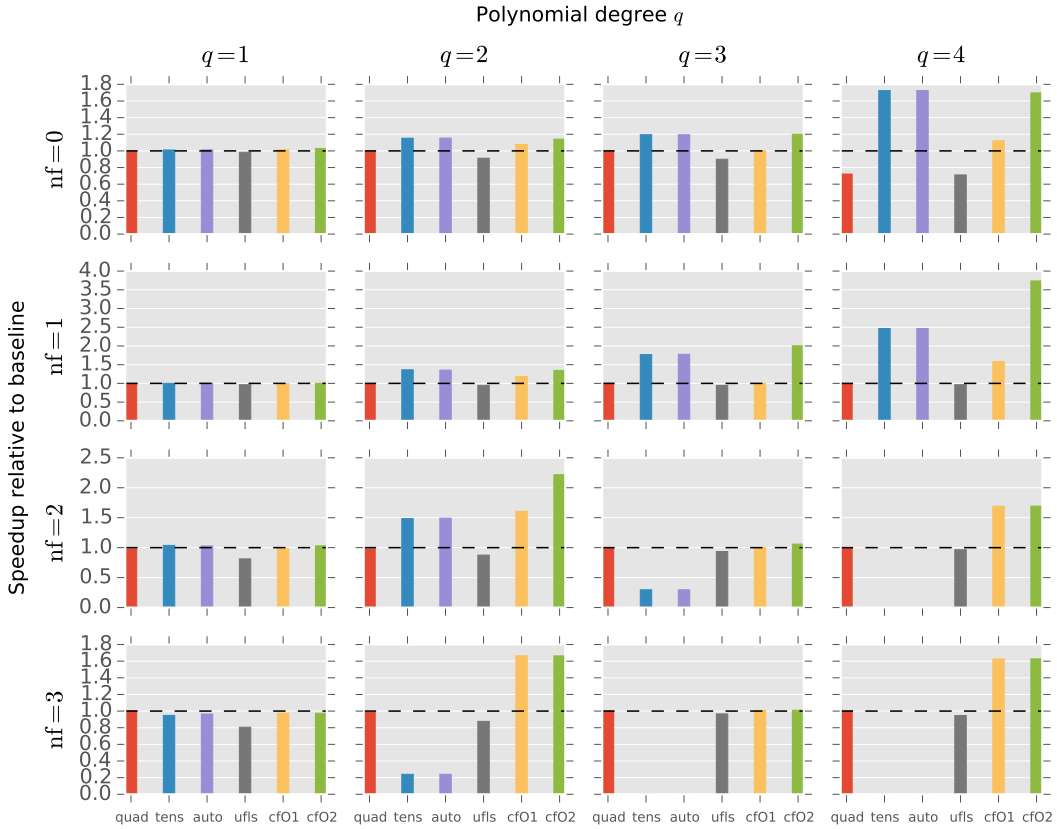


Figure 4.9: Performance evaluation for the *mass* matrix. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

- 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)

4.9.2 Performance Results

We report the results of our experiments in Figures 4.9, 4.10, 4.11, and 4.12 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

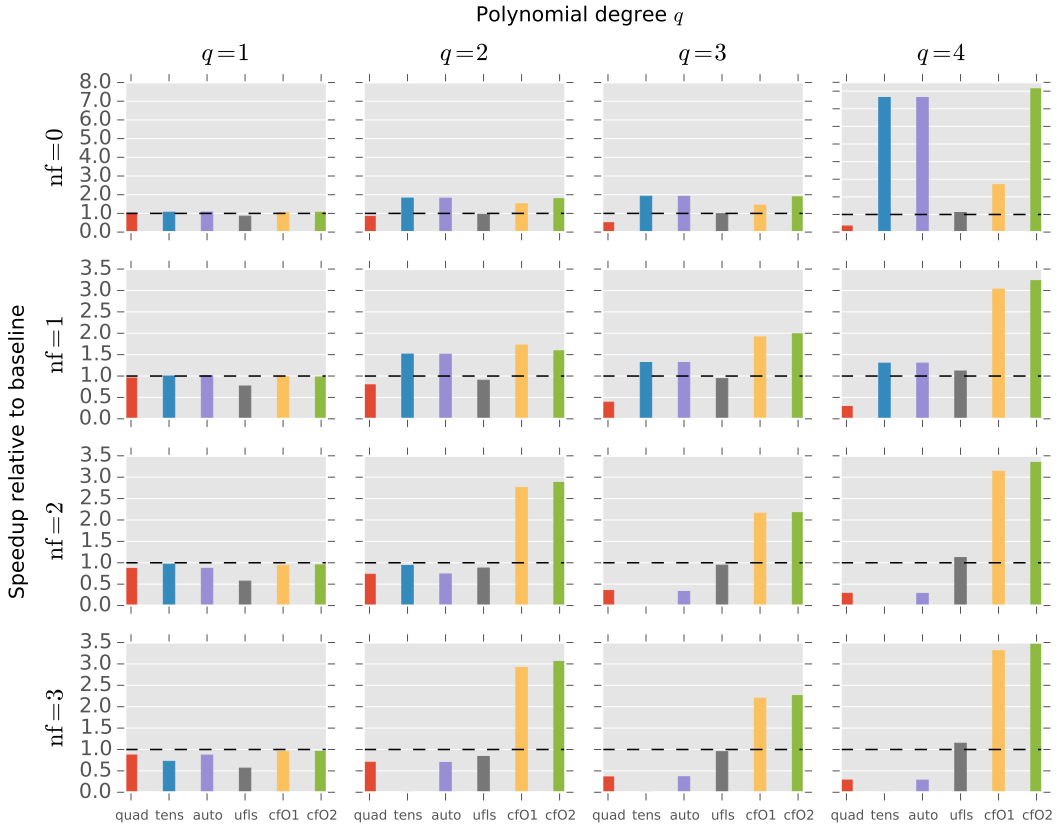


Figure 4.10: Performance evaluation for the bilinear form of a *Helmholtz* equation. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

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

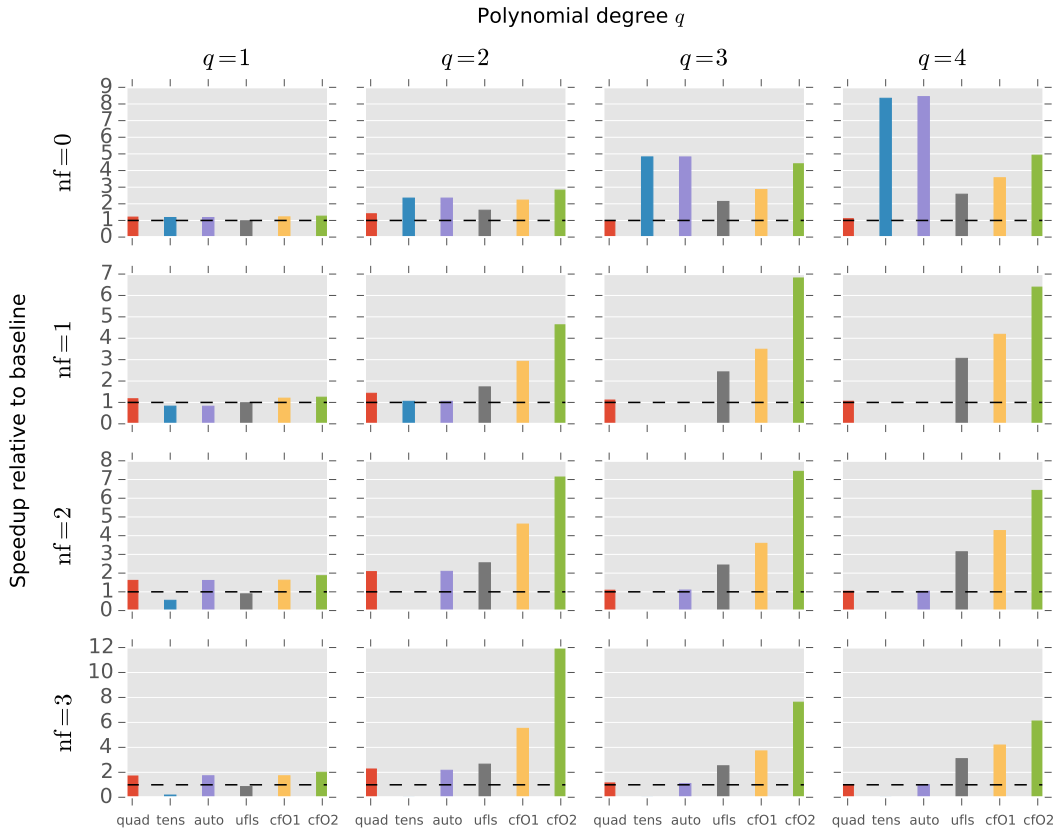


Figure 4.11: Performance evaluation for the bilinear form arising in an *elastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

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.

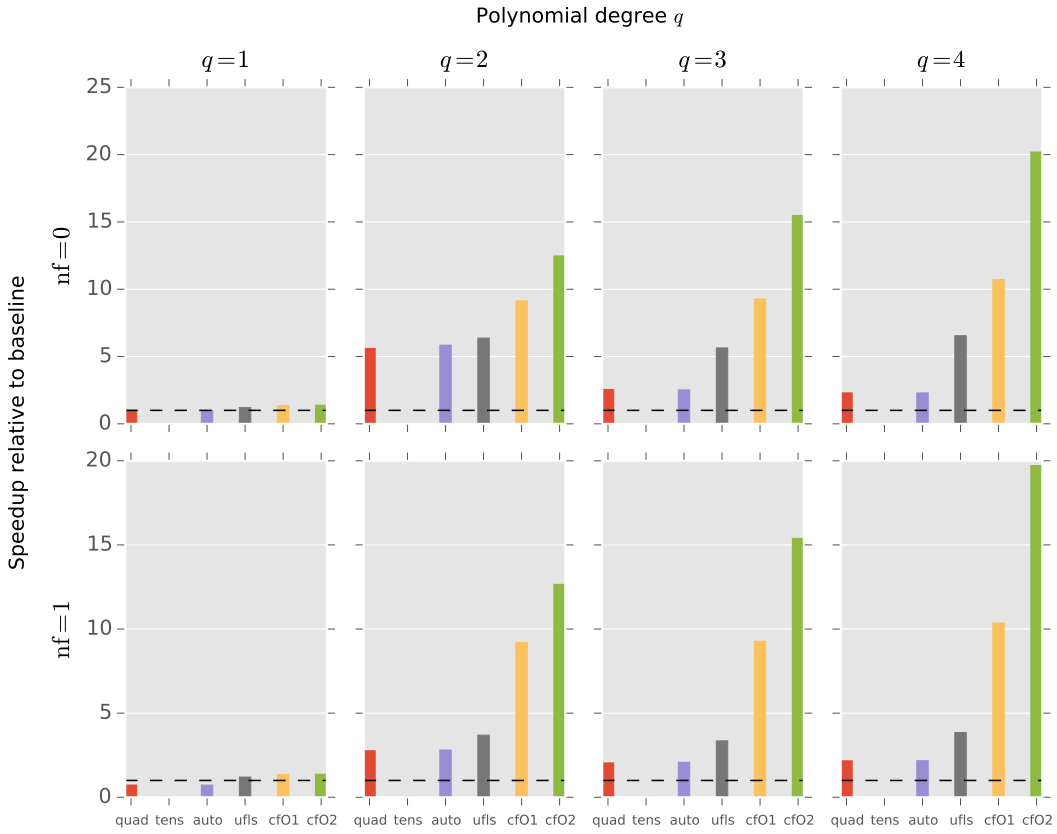


Figure 4.12: Performance evaluation for the bilinear form arising in a *hyperelastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

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[8]. The biggest exception is `quad`, due to the presence of indirection 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 Larsen and Amarasinghe [2000] is often triggered. In `ufls`, `cf01`, and `cf02` the iteration spaces have identical structure, with loop vectorization being regularly applied.

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

Mass matrix We start with the simplest of the bilinear forms investigated, the mass matrix. Results are in Figure 4.9. 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 $T_H = \text{size}(\text{L3})$. 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 par-

allel, 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 4.11. 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 ($5\times$ 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 4.8.3.
- 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 4.12, `cf02` exhibits the largest gains out of all problem instances considered in this section. 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.

4.10 Conclusions

We have developed a theory for the optimization of finite element integration loop nests. The chapter 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 chapter 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.

4.11 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 chapter. 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 so-

phisticated 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

$$A_{jk} = \sum_{\substack{x \in B' \subseteq B \\ y \in B'' \subseteq B}} x_j \cdot y_k \qquad j,k = 0,...,3$$

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.

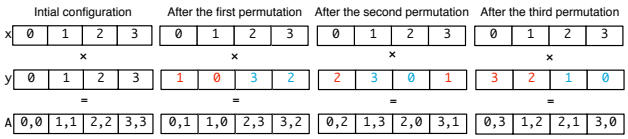


Figure 5.1: Outer-product vectorization by permuting values in a vector register.

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.

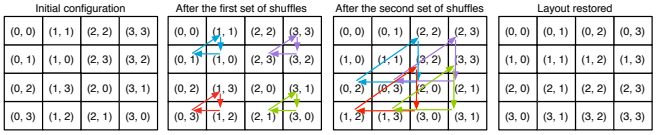


Figure 5.2: Restoring the storage layout after *op-vect*. The figure shows how 4×4 elements in the top-left block of the element matrix A can be moved to their correct positions. Each rotation, represented by a group of three same-colored arrows, is implemented by a single shuffle intrinsic.

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 $q = 4$.

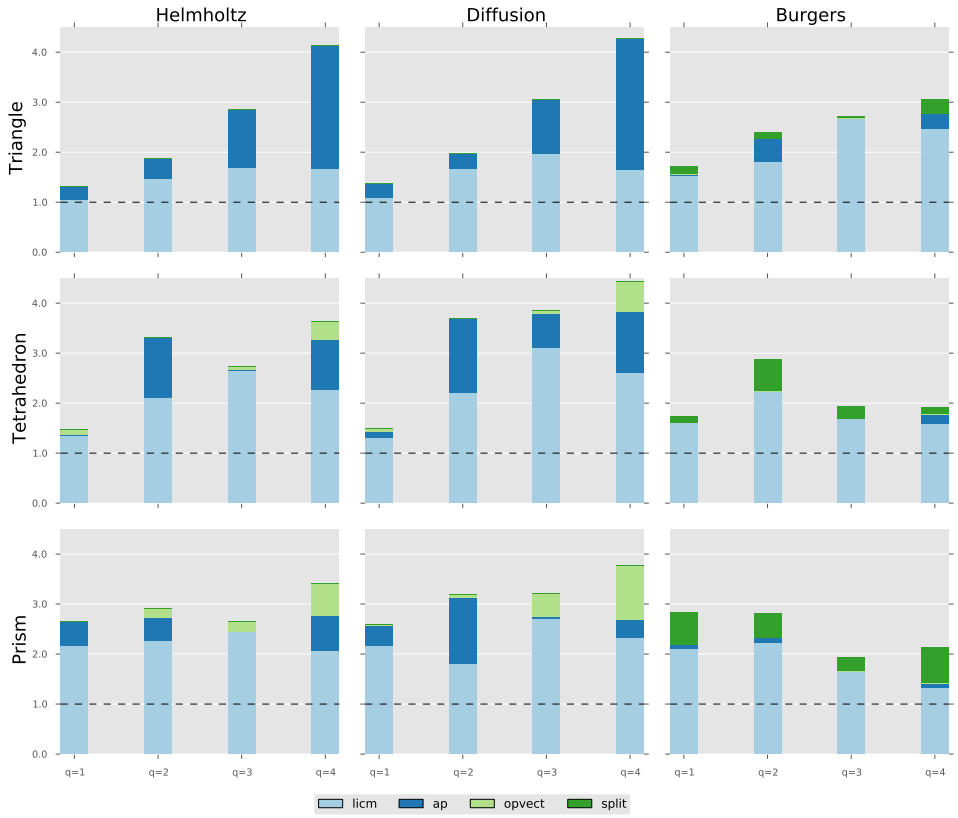


Figure 5.3: Performance improvement due to generalized loop-invariant code motion (*licm*), data alignment and padding (*ap*), outer-product vectorization (*op-vect*), and expression splitting (*split*) over the original non-optimized code. In each plot, the horizontal axis reports speed ups, whereas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

$$\forall_{i,j} \quad A_{ij}^K = \sum_{q=1}^{n_1} \sum_{k=1}^{n_2} \alpha_{k,q}(a', b', c', ...) \beta_{q,i,j}(a, b, c, d, ...) \gamma_q(w_K, z_K) \tag{5.1}$$

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 **?**). In this context, we highlight three notable points.
    1. In Equation (5.1), 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 (5.1), 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 5.1, 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.

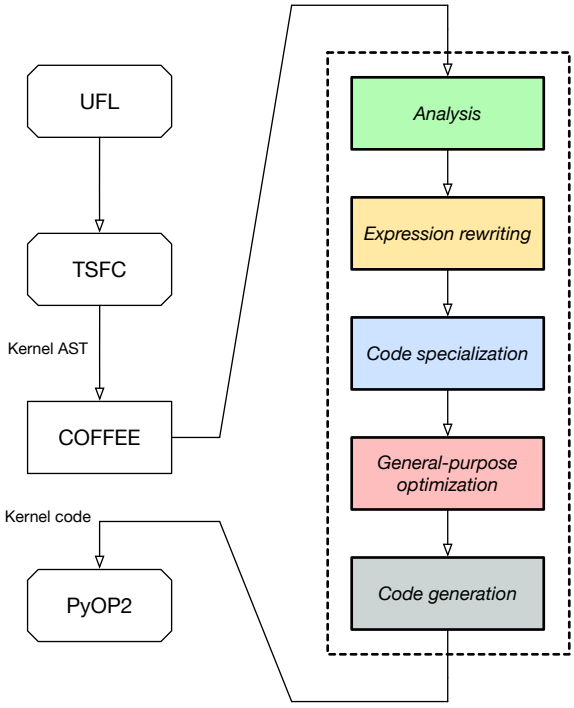


Figure 6.1: The structure of COFFEE and its place within Firedrake.

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 con-

sists 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.

[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.

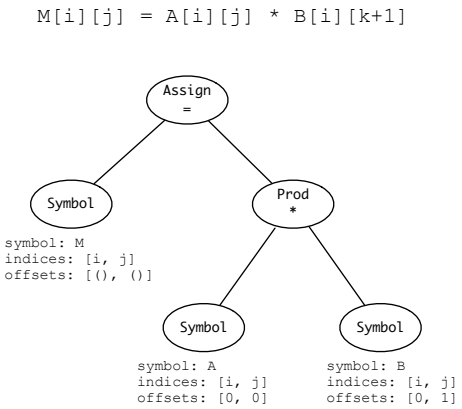


Figure 6.2: AST representation of a C assignment in COFFEE.

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 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 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.

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

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 **?**.

[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.

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 `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

Tensor contraction engine webpage.

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.

Martin Sandve Alnæs. Uflacs - ufl analyser and compiler system. https://bitbucket.org/fenics-project/uflacs, 2016.

Martin Sandve Alnæs, Anders Logg, Garth Wells, Lawrence Mitchell, Marie E. Rognes, Miklós Homolya, Aslak Bergersen, David A. Ham, Johannes Ring, chrisrichardson, and Kent-Andre Mardal. ufl: The unified form language, April 2016. URL http://dx.doi.org/10.5281/zenodo.49282.

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.

Utkarsh Ayachit. *The ParaView Guide (Full Color Version): A Parallel Visualization Application*. Kitware, Incorporated, paraview 4.3 edition, January 2015. ISBN 1930934300. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1930934300.

David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4): 345–420, December 1994. ISSN 0360-0300. doi: 10.1145/197405.197406. URL http://doi.acm.org/10.1145/197405.197406.

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2015. URL http://www.mcs.anl.gov/petsc.

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.

Tobias Brandvik and Graham Pullan. Sblock: A framework for efficient stencil-based pde solvers on multi-core platforms. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1181–1188, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4108-2. doi: 10.1109/CIT.2010.214. URL http://dx.doi.org/10.1109/CIT.2010.214.

S. Brenner and R. Scott. *The Mathematical Theory of Finite Element Methods*. Texts in Applied Mathematics. Springer New York, 2007. ISBN 9780387759333. URL https://books.google.co.uk/books?id=ci4c_R0WKYYC.

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.

Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4385-7. doi: 10.1109/IPDPS.2011.70. URL http://dx.doi.org/10.1109/IPDPS.2011.70.

Alain Darte. On the complexity of loop fusion. *Parallel Comput.*, 26(9): 1175–1193, August 2000. ISSN 0167-8191. doi: 10.1016/S0167-8191(00)00034-X. URL http://dx.doi.org/10.1016/S0167-8191(00)00034-X.

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL http://dl.acm.org/citation.cfm?id=1413370.1413375.

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.

Firedrake. petsc4py: The python interface to petsc, April 2016. URL http://dx.doi.org/10.5281/zenodo.49283.

Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 315–326, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065048. URL http://doi.acm.org/10.1145/1065010.1065048.

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.

Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544160. URL http://doi.acm.org/10.1145/2544137.2544160.

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.

Albert Hartono, Qingda Lu, Xiaoyang Gao, Sriram Krishnamoorthy, Marcel Nooijen, Gerald Baumgartner, David E. Bernholdt, Venkatesh Choppella, Russell M. Pitzer, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Identifying cost-effective common subexpressions to reduce operation count in tensor contraction evaluations. In *Proceedings of the 6th International Conference on Computational Science - Volume Part I*, ICCS'06, pages 267–275, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-34379-2, 978-3-540-34379-0. doi: 10.1007/11758501_39. URL http://dx.doi.org/10.1007/11758501_39.

John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728.

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.

Miklós Homolya and Lawrence Mitchell. Tsfc - two-stage form compiler. https://github.com/firedrakeproject/tsfc, 2016.

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

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

Robert C. Kirby and Anders Logg. Efficient compilation of a class of variational forms. *ACM Trans. Math. Softw.*, 33(3), August 2007. ISSN 0098-3500. doi: 10.1145/1268769.1268771. URL http://doi.acm.org/10.1145/1268769.1268771.

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 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.

Chi-Chung Lam, Thomas Rauber, Gerald Baumgartner, Daniel Cociorva, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. *Comput. Lang. Syst. Struct.*, 37(2):63–75, July 2011. ISSN 1477-8424. doi: 10.1016/j.cl.2010.09.003. URL http://dx.doi.org/10.1016/j.cl.2010.09.003.

Michael Lange, Lawrence Mitchell, Matthew G. Knepley, and Gerard J. Gorman. Efficient mesh management in firedrake using petsc-dmplex. *CoRR*, abs/1506.07749, 2015. URL http://arxiv.org/abs/1506.07749.

Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349320. URL http://doi.acm.org/10.1145/349299.349320.

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.

Anders Logg, Martin Sandve Alnæs, Marie E. Rognes, Garth Wells, Johannes Ring, Lawrence Mitchell, Johan Hake, Miklós Homolya, Florian Rathgeber, Fabio Luporini, Graham Markall, Aslak Bergersen, Lizao Li, David A. Ham, Kent-Andre Mardal, Jan Blechta, Gheorghe-Teodor Bercea, Tuomas Airaksinen, Nico Schlömer, Hans Petter Langtangen, Colin J Cotter, Ola Skavhaug, Thomas Hisch, mliertzer, Joachim B Haga,

and Andrew T. T. McRae. ffc: The fenics form compiler, April 2016. URL http://dx.doi.org/10.5281/zenodo.49276.

Qingda Lu, Xiaoyang Gao, Sriram Krishnamoorthy, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan. Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions. *J. Parallel Distrib. Comput.*, 72(3):338–352, March 2012. ISSN 0743-7315. doi: 10.1016/j.jpdc.2011.09.006. URL http://dx.doi.org/10.1016/j.jpdc.2011.09.006.

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.

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.

Fabio Luporini, Lawrence Mitchell, Miklós Homolya, Florian Rathgeber, David A. Ham, Michael Lange, Graham Markall, and Francis Russell. Coffee: A compiler for fast expression evaluation, April 2016. URL http://dx.doi.org/10.5281/zenodo.49279.

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.

Lawrence Mitchell, David A. Ham, Florian Rathgeber, Miklós Homolya, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Michael Lange, Colin J Cotter, Christian T. Jacobs, Fabio Luporini, Simon Wolfgang Funke, Henrik Büsing, Tuomas Kärnä, Anna Kalogirou, Hannah Rittich, Eike Hermann Mueller, Stephan Kramer, Graham Markall, Patrick E. Farrell, Geordie McBain, and Asbjørn Nilsen Riseth. firedrake: an automated finite element system, April 2016. URL http://dx.doi.org/10.5281/zenodo.49284.

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

Kristian B. Olgaard and Garth N. Wells. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Softw.*, 37(1):8:1–8:23, January 2010. ISSN 0098-3500. doi: 10.1145/1644001.1644009. URL http://doi.acm.org/10.1145/1644001.1644009.

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

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

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.

Florian Rathgeber. *Productive and efficient computational science through domain-specific abstractions*. PhD thesis, Imperial College London, 2014.

Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: automating the finite element method by composing abstractions. *CoRR*, abs/1501.01809, 2015. URL http://arxiv.org/abs/1501.01809.

Florian Rathgeber, Fabio Luporini, and Lawrence Mitchell. firedrake-bench: firedrake bench optimality paper release, April 2016a. URL http://dx.doi.org/10.5281/zenodo.49290.

Florian Rathgeber, Lawrence Mitchell, Fabio Luporini, Graham Markall, David A. Ham, Gheorghe-Teodor Bercea, Miklós Homolya, Andrew T. T. McRae, Hector Dearman, Christian T. Jacobs, gbts, Simon Wolfgang Funke, Kaho Sato, and Francis Russell. PyOP2: Framework for performance-portable parallel computations on unstructured meshes, April 2016b. URL http://dx.doi.org/10.5281/zenodo.49281.

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.

Marie E. Rognes, Anders Logg, David A. Ham, Miklós Homolya, Nico Schlömer, Jan Blechta, Andrew T. T. McRae, Aslak Bergersen, Colin J Cotter, Johannes Ring, and Lawrence Mitchell. fiat: The finite element automated tabulator, April 2016. URL http://dx.doi.org/10.5281/zenodo.49280.

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

ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2504565. URL http://doi.acm.org/10.1145/2503210.2504565.

Francis P. Russell and Paul H. J. Kelly. Optimized code generation for finite element local assembly using symbolic manipulation. *ACM Transactions on Mathematical Software*, 39(4), 2013.

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

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

Barry Smith, Satish Balay, Matthew Knepley, Jed Brown, Lois Curfman McInnes, Hong Zhang, Peter Brune, sarich, stefanozampini, Dmitry Karpeyev, Lisandro Dalcin, tisaac, markadams, Victor Minden, VictorEijkhout, vijaysm, Karl Rupp, Fande Kong, and SurtaiHan. petsc: Portable, Extensible Toolkit for Scientific Computation, April 2016. URL http://dx.doi.org/10.5281/zenodo.49285.

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.

Didem Unat, Jun Zhou, Yifeng Cui, Scott B. Baden, and Xing Cai. Accelerating a 3d finite-difference earthquake simulation with a c-to-cuda translator. *Computing in Science and Engg.*, 14(3):48–59, May 2012. ISSN 1521-9615. doi: 10.1109/MCSE.2012.44. URL http://dx.doi.org/10.1109/MCSE.2012.44.

Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA, 2014.

ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544141. URL http://doi.acm.org/10.1145/2544137.2544141.

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.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL http://doi.acm.org/10.1145/1498765.1498785.

Yongpeng Zhang and Frank Mueller. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 155–164, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259037. URL http://doi.acm.org/10.1145/2259016.2259037.

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.