Imperial College London

Department of Computing

Bho

Fabio Luporini

September 2014

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

List of Tables

List of Figures

x

Chapter 1

Introduction

1.1 Thesis Statement

Computational efficiency in real-world unstructured-mesh-based applications requires specialized sequential code and execution models optimized for non-affine loops. We advocate and demonstrate that an approach centered on domain-specific languages, automated code generation and generalized (i.e. computation-independent) techniques for data locality and parallelism allows achieving this goal and scientists abstracting from the performance optimization problem, which is the key to software productivity.

1.2 Overview and Motivation

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 (1st and 2nd year)

Besides developing novel compilers theory, contributions of this thesis come in the form of tools that have been, and currently are, used to accelerate scientific computations, namely:

- **A run-time library, which can be regarded as a prototype compiler, capable of optimizing sequences of non-affine loops by fusion and automatic parallelization. The library has been used to speed up a real application for tsunami simulations as well as other representative benchmarks.**

  One limiting factor to the performance of unstructured mesh applications is imposed by the need for indirect memory accesses (e.g. `A[B[i]]`) to read and write data associated with the various entities of the discretized domain. For example, when executing a kernel that numerically computes an integral over a cell, which is common in a finite element method, it may be necessary to read the coordinates of the adjacent vertices; this is achieved by using a suitable indirec-

tion array, often referred to as "map", that connects cells to vertices. The problem with indirect memory accesses is that they break several hardware and compiler optimizations, including prefetching and standard loop blocking (or tiling). A first contribution of this thesis is the formalization and development of a technique, called "generalized sparse tiling", that aims at increasing data locality by fusing loops in which indirections are used. The challenges are 1) to determine if and how a sequence of loops can be fused, and 2) doing it efficiently, since the fusability analysis must be performed at run-time, once the maps are available (i.e. once the mesh topology has been read from disk).

- **A fully-operating compiler, named COFFEE[1], capable of optimizing numerical kernels solving partial differential equations using the finite element method. The compiler is integrated with the Firedrake system (Firedrake contributors [2014]).**

  The second contribution, in the context of the finite element method, is about optimizing the computation of so called local element matrices, which can be responsible for a significant fraction of the overall computation run-time. This is a well-known problem, and several studies can be found in the literature, among which [Russell and Kelly, 2013], [Olgaard and Wells, 2010], [Knepley and Terrel, 2013], [Kirby et al., 2005]. With respect to these studies, we propose a novel set of composable, model-aware code transformations explicitly targeting, for the first time, instruction-level parallelism - with emphasis on SIMD vectorization - and register locality. These transformations are automated in COFFEE.

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]COFFEE stands for COmpiler For FinitE Element local assembly.

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), a compiler presented in Chapter 3.

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

1.5 3rd Year Plan

Research plan   I believe the material presented in this report demonstrates that the research conducted so far already represents the bulk of the thesis. The main goal of the third year will be a systematic study of loop fusion for sequences of possibly non-affine loops. This will probably result in the re-implementation of part of the PyOP2 run-time support. Questions and research problems I will address are:

- When is it feasible to fuse loops?

- When is it useful to fuse loops, performance-wise?

- How to automate loop fusion?

This will require a formalization and an implementation efforts that will probably cover the entire third year of studies.

Teaching plan   The teaching plan for the 3rd year is:

- Delivering 3 or 4 hours of new teaching materials (i.e. lectures that I personally prepared) in Paul Kelly's Advanced Computer Architecture course.

- Finishing the PGCert course that I started in my 2nd year. This requires attending and writing reports for two seminars in the coming Autumn term.

- I will help with various 2nd year labs, as I did over the last two years.

1.6 Thesis Outline

The plan is to structure the thesis as follows:

- Chapter 1: **Introduction**. This is going to be similar to the introduction presented in this LSA report.

- Chapter 2: **Background**. The basis for understanding the contributions of the thesis will be provided. This chapter will be composed of three main sections:

    - a summary of the finite element method, which is required to understand the structure of the numerical kernels optimized by COFFEE;

    - an overview of the domain-specific languages, frameworks, methodologies, and compilers that inspired the research;

    - a summary of contemporary multi-core architectures and compilers, including those used for the performance analysis carried out in this thesis.

- Chapter 3: **Generalized Sparse Tiling with the OP2 Loop Chain Abstraction**. First contribution of the thesis. Presented in Chapter 2 of this report.

- Chapter 4: **Generalized Loop Fusion with the OP2 Loop Chain Abstraction.** Second contribution of the thesis. Part of the plan for the third year (see Section 1.5).

- Chapter 5: **Cross-loop Optimization of Arithmetic Intensity for Finite Element Local Assembly**. Third contribution of the thesis. Presented in Chapter 3 of this report.

- Chapter 6: **Conclusions**.

Note that performance evaluations of the techniques presented in Chapters 3, 4, and 5 will be included directly in these chapters.

Chapter 2

Generalized Sparse Tiling with the OP2 Loop Chain Abstraction

In this chapter, we limit ourselves to show a copy of the paper that I managed to publish [Strout et al., 2014]. For the thesis, this chapter will properly be revisited and be more oriented to the OP2's loop chain abstraction. I will also: reuse picture from the ESA reports; include a proper description of algorithms to keep data dependencies under control (algorithms already implemented, but to be formalized yet); show performance results for a real application.

The paper shown in this section has been mainly written by myself and Dr. Michelle Strout. The OP2 part, theory and implementation, is the result of my research activity. We emphasize that this chapter only marginally touches the OP2 part, which will instead be elaborated in the thesis. The other co-authors and their contributions are: Dr. Christopher Krieger (contribution to the general sparse tiling algorithm and Jacobi results), Dr. Carlo Bertolli (contribution to the design of the OP2 sparse tiling algorithm, Mr. Gheorghe-Teodor Bercea (contribution to the implementation of the coloring algorithm based on the K-reachability relation between partitions), Prof. Paul Kelly and Prof. J. Ramanujam (feedback on the whole project).

2.1 Introduction

Intranode parallelization is a difficult problem that many libraries and programming models attempt to address [Amarasinghe et al., 2011]. To expose parallelism in these programming models, scientific simulations commonly express the application as a series of data parallel or reduction loops. However, poor and unpredictable data locality often limits performance and data reuse among the loops is not effectively turned into data locality. This is particularly true for irregular applications that access data using an indirection array, such as `A[B[i]]`. These irregular accesses are common in such fields as computational fluid dynamics, molecular dynamics, differential equation solvers on unstructured meshes, and sparse linear algebra.

Sparse tiling techniques were developed to group iterations of irregular applications into atomic tiles at runtime with an inspector [Douglas et al., 2000, Strout et al., 2004, 2003, Mohiyuddin et al., 2009]. In general, the inspector iterates over index arrays that do not change during the main computation to determine data reorderings or new schedules, like sparse tiling schedules. The resulting tiles have either an implicit or explicit partial ordering (i.e., a task graph) that exposes asynchronous parallelism [Adams and Demmel, 1999, Douglas et al., 2000, Strout et al., 2002]. These benchmark-specific, sparse tiling executors exhibited performance improvements for sparse stencil computations, Gauss-Seidel [Adams and Demmel, 1999, Strout et al., 2004], moldyn [Strout et al., 2003], and sparse matrix powers kernel [Mohiyuddin et al., 2009]. The issue with these approaches is that the sparse tiling inspector algorithms were being developed by hand, per application.

In this paper, we present a generalized, full sparse tiling algorithm that leverages a common pattern in irregular computations and many other scientific codes: a series of parallel and/or reduction loops that reuse data. Previous work introduces an abstraction called the *loop chain*[Krieger et al., 2013] to represent such loop sequences and the data access information about each of the loops. Fig. 2.1 illustrates an example extracted from an unstructured mesh, computational fluid dynamics (CFD) program written using the OP2 **?** library, where it is possible to derive a loop chain abstraction. In the example, the first loop iterates over edges in a mesh, reads data associated with each edge that is stored in the `x` array, and then indirectly

```
 1  void kernel1 (double * x, double * v1, double * v2) {
 2    *v1 += *x;  *v2 += *x;
 3  }
 4  // loop over edges
 5  op_par_loop (edges, kernel1,
 6    op_arg_dat (x, -1, OP_ID, OP_READ),
 7    op_arg_dat (vert, 0, edges2vertices, OP_INC),
 8    op_arg_dat (vert, 1, edges2vertices, OP_INC))
 9
10  // loop over cells
11  op_par_loop (cells, kernel2,
12    op_arg_dat (vert, 0, cells2vertices, OP_INC),
13    op_arg_dat (vert, 1, cells2vertices, OP_INC),
14    op_arg_dat (vert, 2, cells2vertices, OP_INC),
15    op_arg_dat (res, -1, OP_ID, OP_READ))
16
17  // loop over edges
18  op_par_loop (edges, kernel3,
19    op_arg_dat (vert, 0, edges2vertices, OP_INC),
20    op_arg_dat (vert, 1, edges2vertices, OP_INC))
```

Figure 2.1: Section of an OP2 program that is used as a running example to illustrate the loop chain abstraction and show how the sparse tiling algorithm works. The definition of the toy kernel1 shows how all kernels receive their input from the **op_par_loop** implementation.

updates the value of the **vert** data associated with the vertices adjacent to each edge using the **edges2vertices** indirection array. The second loop iterates over cells/triangles and updates all data associated with vertices adjacent to each cell. Finally the third loop visits the edges again. Each of these loops is reusing the data associated with vertices in the unstructured mesh and the data access patterns for each loop can be determined by using information about the OP2 library semantics. Fig. 2.2 shows a possible loop chain with data access edges that are determined at inspector time.

Generalized full sparse tiling (or gFST) converts data reuse within loop chains into data locality, while exposing task-graph parallelism. Fig. 2.3 illustrates a full sparse tiling on the example loop chain in Fig. 2.2. Note that the resulting task graph in Fig. 2.3 has two tiles/tasks that can be executed in parallel, Tiles 2 and 3. Larger examples result in significant improvements in data locality while still exposing sufficient parallelism.

To prototype usage of the loop chaining abstraction as input for a generalized full sparse tiling algorithm, we developed a library where a programmer can replace a sequence of loops with function calls that specify the computation as a loop chain. In Fig. 2.1, the calls to **op_par_loop** can be replaced

with calls to routines that indicate what loops are in the loop chain and for each loop how each iteration in the loop chain accesses data. At runtime this specification is passed to the gFST inspector, which appends the specification with a task graph and mapping of iterations in the loops to tiles/tasks in that task graph. The executor, which replaces the original computation, then executes that task graph. The ultimate goal is to have a compiler identify loop chains within a program, do a cost-benefit analysis to determine whether sparse tiling would be beneficial, and insert inspectors and executors that perform sparse tiling.

The performance benefits of sparse tiling [Douglas et al., 2000, Adams and Demmel, 1999] and specifically full sparse tiling [Strout et al., 2004, 2003, Mohiyuddin et al., 2009] for irregular applications such as Gauss-Seidel, moldyn, and sparse matrix powers kernel have already been shown. Generalizing the full sparse tiling algorithm does increase the inspector overhead, but the improvements in the executor are similar to specialized full sparse tiling executors. To further explore the performance benefits of full sparse tiling, we sparse tiled Jacobi on sparse matrices from the Davis Florida collection [Davis and Hu, 2011] and an Airfoil simulation written in OP2. For the OP2 code, we adapted the current parallelization algorithm to perform a generalized full sparse tiling parallelization. We compared the performance of these benchmarks when parallelized using OpenMP `parallel for` pragmas on each loop versus the performance when the loops were full sparse tiled. The runtime reduction varied from 7% to 47%.

This paper's major contributions are as follows:

- A general full sparse tiling algorithm that applies to any sequence of loops that can be expressed using the loop chain abstraction.

- Performance evaluations of the full sparse tiling inspector/executor strategy when applied to a Jacobi sparse matrix solver and the Airfoil computational fluid dynamics simulation.

We identify a number of important issues arise when generalizing full sparse tiling, and we provide explanations as to how the general algorithm deals with these issues. Additionally, we describe how the key gFST algorithm concepts can be adapted for use in the OP2 implementation context.
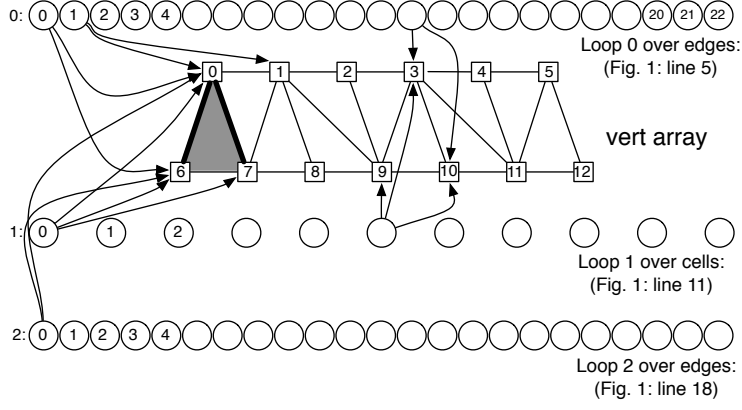
Figure 2.2: Visualization of a possible instance of the loop chain for the sequence of loops in Fig. 2.1. The edge and cell loops use index arrays to indirectly access the data associated with the vertices in the mesh. Squares represent data associated with vertices in a mesh. Circles represent loop iterations. Note that iteration 0 in loop 0 visits the edge connecting vertices 0 and 6 and accesses the data associated with those vertices. Iteration 0 in loop 1 accesses all the vertices associated with the shaded triangular cell. Iteration 0 in loop 2 accesses vertices 0 and 6.

In Section 2.2, we describe how data dependence relations can be derived from the loop chain abstraction and indicate issues related to these dependences that a general sparse tiling algorithm must handle. These issues are addressed by the general full sparse tiling algorithm described in Section 2.3. Section 2.4 describes adapting the OP2 parallelization algorithm to implement generalized full sparse tiling. Section 2.5 reports performance results, Section 2.6 covers work related to full sparse tiling, and Section 2.7 concludes the paper.

2.2 Loop Chains for Generality

Sparse tiling techniques have been developed to improve the data locality within groups of iterations from different loops that share data and, therefore, the performance of the overall computation. *Inspector/executor strategies* implement sparse tiling, where the executor is the transformed code with an added tiling loop and the inspector is a new piece of code that visits index arrays at runtime to determine how iterations can be legally

11
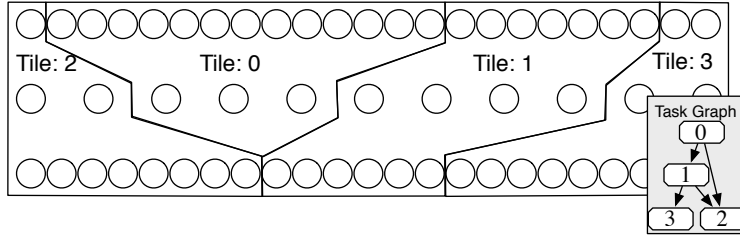
Figure 2.3: A sparse tiling for the loop chain in Fig. 2.2. The iterations in the three loops have been placed into four tiles, which have been partially ordered into a task graph. The partial ordering arises from dependencies between iterations in different tiles.

and profitably grouped into tiles.

This section reviews the loop chain abstraction, describes how data dependences can be derived from a loop chain, and presents issues that a general sparse tiling algorithm of any kind must overcome. The issues are that (1) explicit inspection of the data dependencies between loops to perform sparse tiling more generally is too computationally expensive (Section 2.2.2), (2) when one or more of the loops being sparse tiled are performing a reduction then there needs to be a partial ordering between tiles that perform some reduction operation on the same data element (Section 2.2.3), and (3) when growing tiles to some loop l, it is important to consider the dependencies of loop l on all previous or all subsequent loops (Section 2.2.4). Dependencies in these parallel loops are such that they can be from iteration in any loop L_p to iterations in loop L_q where $p < q$ in general.

2.2.1 Data Dependence Analysis for Loop Chains

The previous sparse tiling approaches cited in Section 2.1 were specialized per benchmark. In this paper we show how the loop chain programming abstraction [Krieger et al., 2013] can be used as a basis for generalized full sparse tiling. 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 provides enough information to compute all of the dependencies in a computation. When the data accesses in the loop chain involve indirect memory accesses like those that are the focus of this paper, then the runtime, or inspector, can determine the data dependen-

cies by querying the data access information contained in the loop chain abstraction.

As described in Krieger et al. Krieger et al. [2013], a loop chain consists of the following:

- L is a sequence of N loops, $L_0, L_1, ..., L_{N-1}$.

- D is a set of disjoint M data spaces, $D_0, D_1, ..., D_{M-1}$.

- $R_{L_l \to D_d}(\vec{i})$ and $W_{L_l \to D_d}(\vec{i})$, where the R and W access relations are defined over for each data space $D_d \in D$ and indicate which data locations in data space D_d an iteration $i \in L_l$ reads from and writes to respectively.

The assumption in a loop chain is that each loop is a *fully parallel loop* or a *reduction loop*. Here a reduction loop is more general than a scalar reduction loop. In a reduction loop each iteration of the loop does a read, an associative and commutative operation, and a write to some element(s) in an array, and multiple iterations could read, modify, write the same data element.

The access relations 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. Loop L_x always comes before loop L_y in the loop chain. The flow dependencies can 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:

$$\{\vec{i} \to \vec{j} \mid \vec{i} \in L_x \wedge \vec{j} \in L_y \wedge W_{L_x \to D_d}(\vec{i}) \cap R_{L_y \to D_d}(\vec{j}) \neq \emptyset\}.$$

Anti and output dependencies are defined as expected.

There are reduction dependencies between two or more iterations of the same loop when those iterations read, modify with a commutative and associative operator, and write to the same location(s). The reduction dependencies in loop L_x are

$$\{\vec{i} \to \vec{j} \mid \vec{i} \in L_x \wedge \vec{j} \in L_x \wedge W_{L_x \to D_d}(\vec{i}) \cap W_{L_x \to D_d}(\vec{j}) \neq \emptyset\}.$$

The reduction dependencies between two iterations within the same loop

indicates that those two iterations must be executed atomically with respect to each other.

2.2.2 Data Dependence Inspection Issue

With the computations that have been sparse tiled in the past (Jacobi, Gauss-Seidel, moldyn, matrix powers kernel), inspecting the dependencies between loops was implemented by traversing index arrays. For example, in Jacobi, each iteration of a loop accesses a set of neighbors. If the sparse matrix is stored in compressed sparse row format then there is a compact list of neighbor identifiers. By iterating over this list the dependence relation is being inspected. The dependence relation can be $\{[i] \rightarrow [j] \mid j \in neighbors(i)\}$. The inspector can traverse the domain for the i iterations and determine data dependencies via the neighbor set.

More generally, traversing data dependencies between loops might require more computation than is preferred in an inspector, which after all must be amortized over multiple iterations of the loop chain. For example, the data dependencies between the first edge loop and the cell loop for the running example in Fig. 2.2 are

$$\{[i] \rightarrow [j] \mid edges2vertices(i, *) = cells2vertices(j, *)\},$$

where $*$ indicates any of the index arrays into vertices for edges or cells. Therefore if an edge and a cell share a vertex, then there is a dependence from the edge iteration to the cell iteration.

Inspecting this relation requires a doubly-nested loop that iterates over all edges and then for each edge all cells, $O(|E||C|)$, where $|E|$ is the number of iterations in the edge loop and $|C|$ is the number of iterations in the cell loop. In Section 2.3, the generalized full sparse tiling algorithm performs tile growth in $O(|E| + |C|)$ instead of $O(|E||C|)$ by avoiding the explicit traversal of data dependences.

Existing inspector/executor techniques for sparse tiling avoid explicitly enumerating data dependencies between loops by being specialized for specific computations. Wavefront parallelization techniques for do across loops also avoid explicit enumeration of data dependencies by tracking how iterations access data [?]. Sparse tiling techniques are different in that they aggregate iterations into tasks and determine a task graph instead of de-

14

termining level sets containing fine grained parallelism. We use a similar approach of tracking data accesses to avoid explicit data dependence enumeration.

2.2.3 Handling Reductions

Strout et al. Strout et al. [2003] sparse tiled the moldyn computation across three of its loops. The loop over interactions between atoms was a reduction loop. Due to reduction dependencies and the resulting computation being performed serially, there was no need to consider dependencies between tiles.

In general, reduction dependencies require that tiles performing a reduction operation to the same data element are given some arbitrary partial order to avoid concurrent execution, which could lead to data races.

2.2.4 Dependencies from Non-Adjacent Loops

Existing sparse tiling techniques only inspect dependencies between adjacent loops. This was because of symmetry between dependencies that caused the dependencies between a loop and much earlier loops to be covered by the transitive closure of dependence steps between intervening loops. In general, a loop could have a dependence, for example an anti dependence, on a loop that is not directly adjacent. The generalized full sparse tiling algorithm handles this case.

2.3 Generalized Full Sparse Tiling Algorithm

To handle all the issues discussed in Section 2.2, the generalized full sparse tiling algorithm uses the data access relations provided by the loop chain abstraction and a data structure we call Ψ that tracks all tiles that write to and read from each data item in each loop to produce a valid execution schedule. Using the given data access relations and the Ψ data structure, the generalized algorithm is able to satisfy the ordering constraints in the loop chain due to data dependencies.

2.3.1 Algorithm Description

The generalized full sparse tiling algorithm takes a loop chain and assigns each iteration of the loops to a tile. Along with the iteration-to-tile mapping,

---
ALGORITHM 1: The Generalized Full Sparse Tiling Algorithm
---

1 **GeneralizedFullSparseTile**
  Input: Loop Chain $LC = (L, D, R, W)$, s, T
  Output: θ, G
  Data: Ψ_*
2 // Initialize all fields of Ψ to top or empty set
3 // Initialize the tiling function θ values to $\top$
4 $\theta(L_s, *) = PartitionSeedSpace(L_s, R, W, T)$
5 UpdateAccessTable(Ψ_*, s)
6 //Tile the loop chain
7 **foreach** L_l *in* L_{s-1} *to* L_0 **do**
8 $\quad$ | $\quad$ BackwardTile$(L, l, R, W, \theta, \Psi_*)$) UpdateAccessTable$(\Psi_*, l)$
9 **end foreach**
10 **foreach** L_l *in* L_{s+1} *to* L_{N-1} **do**
11 $\quad$ | $\quad$ ForwardTile$(L, l, R, W, \theta, \Psi_*)$ UpdateAccessTable(Ψ_*, l)
12 **end foreach**
13 // Partial ordering of tiles
14 G = BuildTaskGraph(L, D, Ψ_*, T) **return** θ, G
---

a task graph representing a partial ordering of the tiles is generated. More precisely, the input to the algorithm is a loop chain (LC), the index of the loop chosen for seed partitioning (s), and the number of tiles (T). Note that any loop can be selected for seed partitioning, but heuristically a loop in the middle of the loop chain results in fewer dependencies between tiles [**?**]. The number of tiles is a tuning parameter used to balance data locality and parallelism. The output is a function θ that maps each iteration of each loop in the chain to a tile and a task graph G that captures the partial ordering among the tiles due to data dependencies.

The general full sparse tiling method for loop chains consists of four phases: *initialization, backward tiling, forward tiling*, and *task graph creation*. Algorithm 1 shows the pseudo-code.

Phase 1: Initialize internal data. Data dependency information is required during task graph creation. Rather than tracking data dependencies directly, which may be prohibitively expensive, the algorithm maintains information pertaining to data reads and writes with respect to tiles. The set of tiles in a particular loop (l) that read a particular data item ($\vec{v}$) in data space (d) is denoted as $\Psi_R(d, \vec{v}, l)$. The tiles that write are tracked in a similar set $\Psi_W(d, \vec{v}, l)$.

Additionally, tile data access information is used during backward and forward tiling. Associated with each of the Ψ_R and Ψ_W sets are single

16

values that record the first and last tiles that access a specific data element. $\Psi_{LR}(d, \vec{v}, l)$ is the last read performed on the vth data element of the data space D_d from loop L_l. Replacing the subscripts with FR, FW, and LW correspond to the first read, first write and last write respectively. The initialization phase initializes all the tile assignments to top ($\top$) and sets all of the data access information in Ψ to empty set or top as appropriate.

The initialization phase also includes a preliminary partitioning of the seed loop's iteration space, L_s. Partitioning the seed loop involves assigning each iteration of the seed loop to a tile. $UpdateAccessTable$ updates the values in all internal state data sets (Ψ_*) with respect to the seed partitioning. Specifically, since there is a tile assignment for all the iterations in the seed loop, it will be possible to determine the set of tiles that read and write to each data element accessed in that seed loop.

For example, Fig. 2.4 shows a portion of the Ψ data structure for the loop chain example in Fig. 2.2. In this example, the seed loop is chosen as loop 1. This is the loop over cells (a cell is a triangle here). In this example we follow the creation of tile 2. After the seed partitioning, iterations 0 and 1 in loop 1 are in tile 2, $\Theta(1,0) = 2$ and $\Theta(1,1) = 2$. The Ψ data structure contains tile access information for each data element (i.e., vertices in the mesh) at each loop in the computation (i.e., note three columns for each vertex, one for each of the three loops). The left side of Fig. 2.4 shows the initial write sets for part of the vert data structure. The first two cells have been put into tile 2 in the seed partition loop. The vertices adjacent to these two cells have a 2 in the center column to illustrate that the tile 2 is in the sets $\Psi_W(vert, 0, 1)$ and $\Psi_W(vert, 1, 1)$. The zeros in the middle columns of the Ψ table are for vertices adjacent to cells that are in tile 0.

Phase 2: Backward Tiling. The result of tiling is that each iteration in the loop chain is assigned to a tile. Backward tiling (see Algorithm 2) starts with the iterations in the seed partitions and grows tiles to earlier loops ensuring that the data dependencies are satisfied. Each iteration in the loop being tiled is assigned to either the existing tile assignment or $\Psi_{FW}(d, \vec{v}, l)$ or $\Psi_{FR}(d, \vec{v}, l)$, depending on which occurs first (the result of MIN).

Consider the running example in Fig. 2.4. Backward tiling in this example only occurs on loop 0. Iteration 0 in loop 0 starts with a tile value of top, $\Theta(0,0) = \top$. Iteration 0 in loop 0 does a reduction operation on two

0: ⓪①②③④⑤⑥⑦⑧ 0: ⓪①②③④⑤⑥⑦⑧

| Tile 2 | | Tile 0 | | | Tile 2 | | Tile 0 | |
| ⓪ | ① | ② | ③ | | ⓪ | ① | ② | ③ |

1: 1:

2: ⓪①②③④⑤⑥⑦⑧ 2: ⓪①②③④⑤⑥⑦⑧

$\Psi_W(vert,0,1) = \{2\}$ $\Psi_W(vert,0,0) = \{0,2\}$

After Initial Partition   **After Backward Tiling**
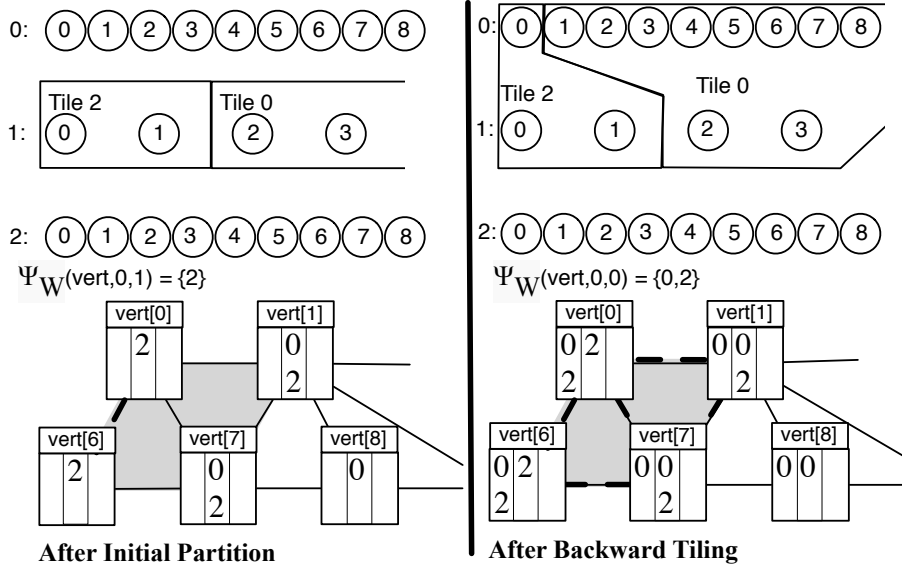
Figure 2.4: This figure illustrates part of the evolution of the Ψ_W data structure for the loop chain example in Fig. 2.2. After the initial seed partitioning of loop 1, which iterates over cells, the middle columns (loop 1) associated with each vertex includes the set of tiles for cells that are adjacent to the vertex. After backward tiling, the Ψ_W data structure is updated to include the tile numbers for all the edges that access a vertex in loop 1. Only one edge is in tile 2 in loop 0, so all vertices shown are written to by edges in Tile 0.

vertices (0 and 6, the edge is represented as a bold, dashed line) and we have $\Psi_{FW}(vert,0,1) = 2$ and $\Psi_{FW}(vert,6,1) = 2$. Therefore, iteration 0 of loop 0 is assigned to tile 2, $\Theta(0,0) = 2$. As another example, iteration 1 of loop 0 accesses the vertices 0 and 1, and $\Theta(0,1) = \top$, $\Psi_{FW}(vert,0,1) = 2$, and $\Psi_{FW}(vert,1,1) = 0$. $MIN(\Theta(0,1), \Psi_{FW}(vert,0,1), \Psi_{FW}(vert,1,1)$ is 0 and therefore iteration 1 in loop 0 is assigned to tile 0, $\Theta(0,1) = 0$. Intuitively any time an earlier loop iteration will be writing to a data item that an iteration in a later loop accesses, then the earlier loop iteration needs to be in the same or earlier tile.

Once the tile assignments are chosen, Ψ_* is updated to reflect the changes. These values are reflected in the first column of each vertex in Fig. 2.4.

Phase 3: Forward Tiling. In this phase, loops after the seed space are

ALGORITHM 2: The Backward Tiling Algorithm

1 **BackwardTile**
  Input: LC, l, Ψ_*
  Output: θ
2 Define: MIN($\top$,X) = X.
3 **foreach** $\vec{i} \in L_l$ **do**
4   // all datasets
5   **foreach** $d \in D$ **do**
6   // all data elements read by this iteration
7   **foreach** $\vec{v} \in R_{L_l \to D_d}(\vec{i})$ **do**
8   // each later loop's access tables
9   **foreach** $L_k \in \{L_{l+1} \ to \ L_s\}$ **do**
10   // anti dependence
11                 $\theta(l,\vec{i}) = MIN(\theta(l,\vec{i}), \Psi_{FW}(d,\vec{v},k))$
12   **end foreach**
13   **end foreach**
14   // all data elements written by this iteration
15   **foreach** $\vec{v} \in W_{L_l \to D_d}(\vec{i})$ **do**
16   **foreach** $L_k \in \{L_{l+1} \ to \ L_s\}$ **do**
17   // output dependence
18                 $\theta(l,\vec{i}) = MIN(\theta(l,\vec{i}), \Psi_{FW}(d,\vec{v},k))$
19   // flow dependence
20                 $\theta(l,\vec{i}) = MIN(\theta(l,\vec{i}), \Psi_{FR}(d,\vec{v},k))$
21   **end foreach**
22   **end foreach**
23   **end foreach**
24 **end foreach**

tiled. This process starts with the iteration space immediately following the seed loop and proceeds forward, loop by loop, until reaching the last loop in the chain. The forward tiling algorithm uses MAX where the backward tiling algorithm uses MIN. It exploits the last read and write information to ensure data dependencies are satisfied.

Phase 4: Task Graph Creation. The edges in the task graph represent the data dependencies that occur between iterations in different tiles (for example see Fig. 2.3). The four steps of the task graph creation, as shown in Algorithm 3, correspond to the four types of dependencies between tiles. To avoid cycles in the task graph, the source tile of an edge should always be numbered lower than the target tile for the edge.

Reduction dependencies are detected when a single entry in the Ψ_W table includes multiple tiles, thus indicating that multiple tiles are writing to a

19

ALGORITHM 3: The task graph is determined by inspecting all tiles that read and write to each data element.

---

1 **BuildTaskGraph**
  Input: L, D, Ψ_*, T
  Output: $G = (V, E)$
2 $E = \emptyset, V = \{0, 1, \cdots, T-1\}$ // Each tile is task
3 **foreach** d *s.t.* D_d *in* D_0 *to* D_{M-1} **do**
4   **foreach** l *s.t.* L_l *in* L_0 *to* L_{N-1} **do**
5   **foreach** $\vec{v} \in D_d$ **do**
6   // Reductions
7             $E = E \cup \{[s] \to [t] \mid s \in \Psi_W(d, \vec{v}, l) \wedge$
8                 $t \in \Psi_W(d, \vec{j}, l) \wedge s < t\}$
9   // Flow dependencies
10   **foreach** k *s.t.* L_k *in* L_l *to* L_0 *until* $\Psi_{LW}(d, \vec{v}, k) \neq \top$ **do**
11                 $E = E \cup \{[s] \to [t] \mid s = \Psi_{LW}(d, \vec{v}, k) \wedge$
12                    $t \in \Psi_R(d, \vec{j}, l) \wedge s < t\}$
13   **end foreach**
14   // Anti dependencies
15   **foreach** k *s.t.* L_k *in* L_l *to* L_{N-1} *until* $\Psi_{FW}(d, \vec{j}, k) \neq \top$ **do**
16                 $E = E \cup \{[s] \to [t] \mid s \in \Psi_R(d, \vec{v}, l) \wedge$
17                    $t = \Psi_{FW}(d, \vec{v}, k) \wedge s < t\}$
18   **end foreach**
19   // Output dependencies
20   **foreach** k *s.t.* L_k *in* L_l *to* L_{N-1} *until* $\Psi_{FW}(d, \vec{v}, k) \neq \top$ **do**
21                 $E = E \cup \{[s] \to [t] \mid s = \Psi_{LW}(d, \vec{v}, l) \wedge$
22                    $t = \Psi_{FW}(d, \vec{v}, k) \wedge s < t\}$
23   **end foreach**
24   **end foreach**
25   **end foreach**
26 **end foreach**

---

single vertex in the same reduction loop. A partial ordering from the lower to higher numbered tiles is placed in the task graph to avoid data races. Edges representing flow dependencies are created by connecting all tiles that read a specific data element within a given loop to the tile that has most recently written to that data element in a previous loop. Anti-dependence edges are similar, but connect all of the tiles that read a given element in a given loop to the first tile in a subsequent loop that writes to the tile. Output dependences are found by connecting the last write of a data element to the first write of the same element in subsequent loops.

The generalized full sparse tiling algorithm maps each iteration in the loop chain to a tile with the tile mapping function θ and generates a partial

ordering between the tiles in the form of a task graph. One possible execution model is to then execute each tile/task serially (loops executed in the original loop sequence within each tile) and to execute tiles/tasks that do not share a partial ordering in parallel.

2.3.2 Algorithm Complexity and Correctness

The backward (and forward) tiling algorithms traverse all iteration and data index pairs in the read $R_{L \to D}$ and write $W_{L \to D}$ relations. For each such pair, the impact of all previous loops in the loop chain on the data item in question are queried in the Ψ data structure (line 10 of the Backward Tiling algorithm). Therefore the complexity is $O(NMP)$ complexity, where N is the total number of iterations in the loop chain, M is the average number of data accesses per iteration, and P is the number of loops in the loop chain. Note that NM is the number of pairs in the read $R_{L \to D}$ and write $W_{L \to D}$ relations. We avoid $O(N^2)$ behavior such as the $O(|E||C|)$ from Section 2.2.2 that would be needed if all data dependencies between iterations were explicitly determined.

We do need to explicitly determine dependencies between tasks to specify the task graph. The task graph construction algorithm has a worst case complexity of $O(T^2)$, where T is the number of tiles, because all of the tiles could read or write to a particular data element. However, if that were the case, then the problem is not sparse enough for full sparse tiling to be effective.

Any schedule for iterations in a loop chain is correct if the schedule satisfies the partial ordering between iterations in the loop chain dictated by the data dependencies detailed in Section 2.2.1. The full sparse tiling algorithm indirectly satisfies these data dependencies without actually having to explicitly construct the data dependencies at runtime. The full proof is omitted due to space.

2.4 OP2 implementation of gFST

OP2[1] is a library for implementing applications that solve partial differential equations over unstructured meshes. For example, OP2 is employed in

---

[1]OP2 denotes that this is the second generation of the OPlus library, or Oxford Parallel Library.

the Hydra CFD application, used at Rolls Royce for the simulation of next-generation components of jet engines, and in Volna [**?**], a CFD application for the modelling of tsunami waves. These applications are composed of a large number of parallel or reduction loops (hundreds in Hydra, tens in Volna). This section describes how we adapted the current OP2 parallelization algorithm to implement generalized full sparse tiling for OP2 programs. Inclusion of the calls to the OP2 gFST inspector and introduction of tile loops into the executor code is done manually. The goal is to assess the performance of gFST in the context of a mature code base like OP2, before incorporating this optimization into the OP2 compiler.

2.4.1 Loop chains in the OP2 Library

OP2 offers abstractions for modeling an unstructured mesh in terms of sets, datasets and mappings between sets. OP2 programs are expressed as sequences of parallel loops, each loop applying a user-programmed function, or "kernel", to every element in the iteration set. For each set element, datasets are accessed through maps. For example, the parallel loops in the program in Fig. 2.1 iterate over sets of mesh elements (`edges`) and access datasets (`vert`) via mappings, which can be indirect (`edges2vertices`) or direct (`op_id`).

In OP2, access modes are used to indicate how datasets are being accessed: (1) `OP_READ` - the dataset is only read, (2) `OP_WRITE` - the dataset is written to, (3) `OP_INC` - an increment for each value of that dataset is computed without reading the actual value of the dataset. An access descriptor (op_arg_dat) contains: a dataset, a mapping and an access mode. Each parallel loop contains an access descriptor for each dataset used by the kernel. This information captures the loop chain abstraction in OP2.

2.4.2 Specializing gFST for OP2 Loop Chains

The standard OP2 OpenMP parallelization is done on a per loop basis and is achieved by block partitioning the iteration set (e.g. cells, edges). In the OP2 gFST inspector, instead, partitioning occurs only once and is performed on the mesh vertices. These partitions are grown to tiles in the backward and forward tiling operations. In both cases, serialization of iterations incrementing the same value (i.e., reduction dependencies) is

enforced by giving a color to partitions and allowing only same-colored partitions to execute in parallel, with iterations inside a partition being executed sequentially.

The OP2 gFST inspector extends this technique by coloring partitions to respect tile-to-tile dependencies exposed by the forward and backward tiling operations. The inspector builds a seed partition graph (SPG) with each node of the SPG being a partition of vertices. Edges are inserted in the SPG based on the K-reachability relation between partitions: if two vertices in separate partitions are within K edges or cells of each other then their corresponding partition nodes in the SPG will be connected. Building the SPG costs $O(TN(B^K))$, where T is the number of tiles, N is the number of vertices on the border of a partition, B is the average out degree for vertices in the mesh, and K is how many edges the depth-first search visit traverses from each border vertex. In our examples B is 3 or 4, K is the number of loops in the loop chain and tends to be small, and N is around 40. The SPG is then colored to prevent same-colored tiles from sharing a dependence.

The gFST algorithm tracks all the tiles that read from and write to a particular data item in *each* loop. This information, stored in the Ψ_* data structure, is used to determine a partial order of tile execution due to data dependencies. The OP2 gFST simplifies dependency tracking by exploiting the fact that vertices are accessible by all iteration set via mappings. Instead of storing all the tiles that read and write to a particular vertex per loop, it is only necessary to track the first tile that reads or increments a vertex while doing backward tiling and the last tile that reads or increments a vertex while doing forward tiling. Specifically the complexity of the OP2 gFST tile growth is $O(NM)$ instead of $O(NMP)$, where where N is the total number of iterations in the loop chain, M is the average number of data accesses per iteration, and P is the number of loops in the loop chain; the loop at Line 4 in Algorithm 3 is unnecessary.

Adapting the existing OP2 parallelization algorithm to perform gFST highlights the most crucial aspects of the algorithm while illustrating that these features can be specialized for particular implementation contexts. The most crucial features are (1) there needs to be some mechanism for providing a seed partitioning, whether on an iteration space or a data space that all iteration spaces access directly or indirectly, (2) flow, anti, and output dependencies need to be respected during backward and forward
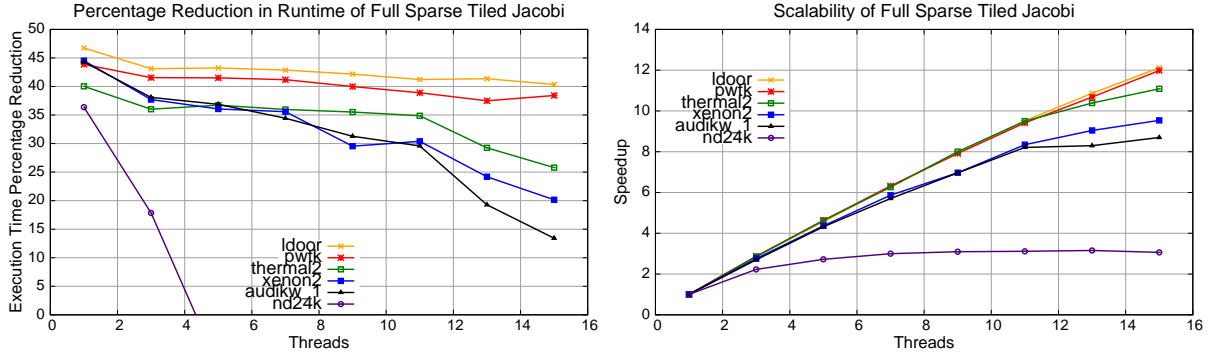
Figure 2.5: The Jacobi solver's loop chain performance in terms of percentage reduction over the simple blocked parallel version, and speedup over the sequential full sparse tiled versions. Results for 6 different sparse matrices are shown. The nd24k line goes off the chart, reaching -152 percentage reduction at 15 cores; full sparse tiling does not always result in better performance.

tile growth to maintain tile atomicity, (3) flow, anti, output and reduction dependencies between tiles need to be partially ordered to avoid data races.

2.5 Evaluation of Generalized Sparse Tiling

To evaluate the performance improvements achieved by full sparse tiling, we conducted several experiments. In all the experiments, the optimal tile size (i.e. the one leading to the best execution time) was determined empirically, for each combination of machine and application. Our objective is to explore the impact of gFST on performance, and to characterize the circumstances where the approach is profitable.

2.5.1 The Sparse Jacobi Benchmark

The first experiment was the full sparse tiling of a Jacobi sparse matrix solver. 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` and OpenMP tasks were used to execute the task graph.

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 testbed 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 [**?**], 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.

Figure **??** shows the execution time reduction achieved by full sparse tiling the Jacobi solver compared with the execution time of a simple blocked parallel version using OpenMP `parallel for` directives on the unrolled loops. 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 full sparse tiled. This matrix is highly connected and yields a task graph that has limited parallelism. 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.

These execution times 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. As the inspector code matures and becomes more efficient, this cost will diminish.

In Figure **??**, the scalability of the full sparse tiled Jacobi solver is shown. In general, speedups of between 8 and 12 times over the single-threaded performance were observed when using 15 threads. A clear outlier is again the nd24k matrix that did not scale past 3.2 times the single thread performance. The high degree of connectivity present in this matrix limited the parallelism available in the task graph, which in turn limited the scalability.
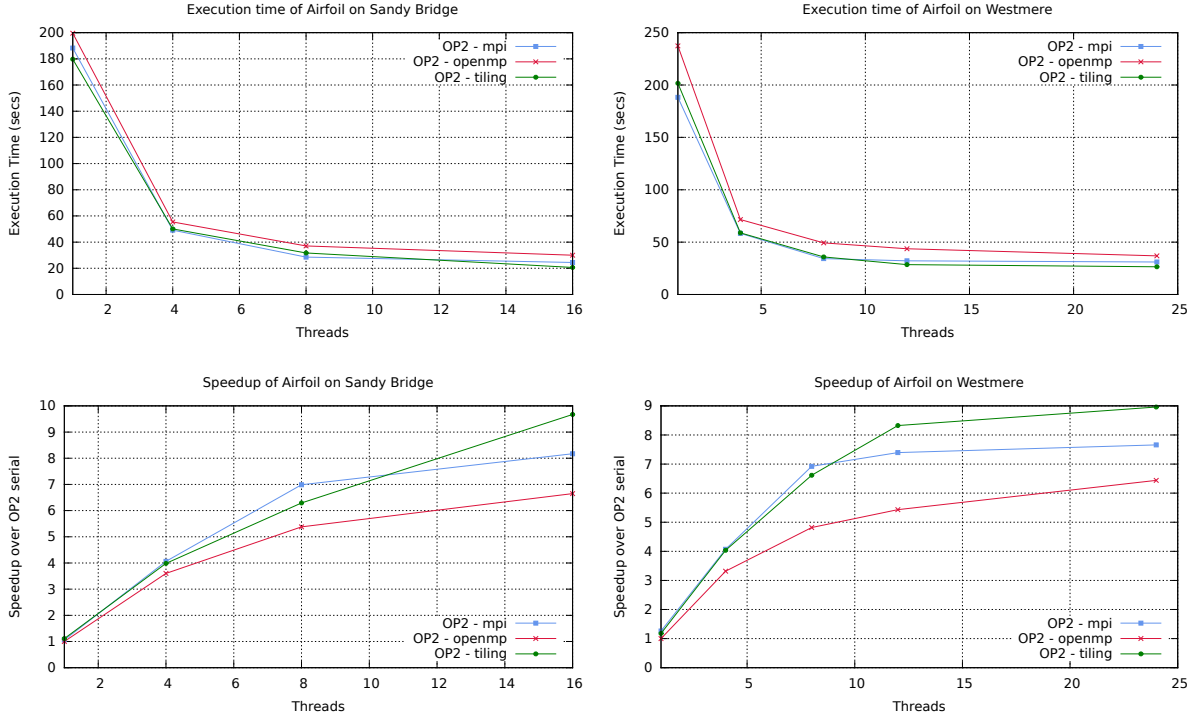
Figure 2.6: The Airfoil's loop chain performance in terms of execution time (in seconds) and speedup relative to the best sequential execution time for Sandy Bridge(a,c) and Westmere(b,d). The speedup is evaluated with respect to the *omp* version with one thread (i.e. the slowest sequential back-end).

2.5.2 OP2 Airfoil Benchmark

The OP2 adaptation of the generalized sparse tiling technique was evaluated in a representative unstructured mesh application called Airfoil [?]. Three implementations of Airfoil, namely *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 a more recent 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 OP2 Airfoil application consists of a main time loop with 2000 iterations. This loop contains a sequence of four parallel loops that carry out the computation. In this sequence, the first two loops, called *adt-calc* and *res-*

calc, constitute the bulk of the computation. *Adt-calc* iterates over cells, reads from adjacent vertices and write to a local dataset, whereas *res-calc* iterates over edges and exploits indirect mappings to vertices and cells for incrementing indirect datasets associated to cells. These loops share datasets associated with cells and vertices. Datasets are composed of doubles.

In the *omp* and *mpi* implementations of Airfoil, the OpenMP and the MPI back-ends of OP2, were used. The effectiveness of these parallelization schemes has been demonstrated in **?**. The OP2 OpenMP back-end has been intuitively described in Section 2.4. The *tiled* implementation exploits the OP2 gFST library for tiling a loop chain composed of 6 loops: the time loop was unrolled by a factor of two so as to tile over *adt-calc* and *res-calc* twice. The OP2 gFST library uses $METIS$ [**?**] for computing a seed partitioning of the mesh vertices.

Figure 2.6 shows the scalability and runtime reduction realized by full sparse tiling the loop chain on the Westmere and Sandy Bridge machines. 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. It is left as further work extending gFST algorithms to work around this issue. 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 the overhead of the inspector. By also including the inspector cost, the aforementioned improvements over *mpi* reduce to roughly 10% on both platforms. However, as the time-marching loop in real-world OP2 applications tends to be larger than in Airfoil, we expect the overhead of the inspector to be, in general, smaller. In addition, we believe the current implementation of the gFST inspector is amenable to several optimisations.

2.5.3 Discussion of the Performance Results

The performance results presented here for Jacobi and Airfoil support previous work demonstrating the benefits of full sparse tiling [Strout et al., 2004, 2003, Mohiyuddin et al., 2009]. Extending this approach by grouping iteration that share data across loops into tiles improves performance due to improved data locality. On multicore machines this avoids memory

bandwidth saturation while scaling.

Insufficient parallelism in the task graph can limit the performance improvements. This is observed with the nd24k sparse matrix in the Jacobi Benchmark. A possible future method for increasing parallelism, when it is limited in the task graph, is to take advantage of the parallelism within each loop in each tile.

An additional limiting factor is inspector overhead. This overhead must be amortized of the full execution. The effect of this is limited in irregular scientific applications because they typically require inspector-time partitioning already.

Choosing the correct input parameters to the tiling process is key to achieving performance improvements. The parameters include, the number of tiles, the iteration space to use as the seed partition, and the numbering of the seed partition. The quality of the seed partition and associated coloring is especially important. Together these determine the degree of parallelism in the task graph.

2.6 Related Work

Our definition of a loop chain was presented in Krieger et al. [2013] along with a discussion of how the loop chain abstraction is complimentary to previous projects that performed task scheduling in order to achieve asynchronous parallelism.

For unstructured codes, there has been various inspector/executor strategies [Salz et al., 1991] that reschedule across loops to improve data locality while still providing parallelism [Douglas et al., 2000, Strout et al., 2002, Demmel et al., 2008, Krieger and Strout, 2012]. These methods include *communication avoiding* approaches [Mohiyuddin et al., 2009] that optimize a series of loops over unstructured meshes. These strategies fall under the broader category of sparse tiling. In this paper we have presented a generalized sparse tiling algorithm, whereas previous work was specific to particular benchmarks.

Various code transformation have been developed to reschedule computation and reorder data for loop-chain-like code patterns. Many of these techniques also generate parallel execution schedules for the loops. The approach in Ravishankar et al. [2012] identifies *partitionable loops*, and sched-

ules these loops for execution on a distributed memory machine. Likewise, there are approaches that take parallel loops identified by OpenMP pragmas and transform them for execution on distributed memory clusters [Basumallik and Eigenmann, 2006].

The approach presented in this paper differs from these techniques in two key ways. First, these approaches generate a schedule in which each partition or processing element executes its assigned iterations of one loop, then communicates a subset of its results to other partitions that are dependent on that data. After executing its iterations of a loop, each processing element potentially waits to receive data from other partitions. The full sparse tiling approach described here does not require any synchronization or communication during the execution of a tile due to the *atomicity* of the tile. Before a tile begins execution, it waits until all necessary data is available and then executes from start to finish without further communication or synchronization. This approach can better exploit the locality available across the sequence of loops.

2.7 Conclusions

Full sparse tiling has previously been shown to deliver significant performance gains when applied ad hoc to specific applications. In this paper, we present a generalized algorithm for correctly sparse tiling any valid loop chain. This algorithm uses the newly developed loop chain abstraction as input, improves inter-loop data locality, and creates a task graph to expose shared-memory parallelism at runtime. For the sparse Jacobi benchmark, we showed that even though the unoptimized, generalized inspector has high overhead, the resulting executor has performance improvements. By adapting the sparse tiling inspector for unstructured mesh applications written using the domain-specific library OP2, we see performance improvements over even MPI on the Airfoil benchmark. These results add to the growing body of evidence that sparse tiling techniques enable communication avoidance and therefore improve parallel performance and scaling on multicore architectures. Future work includes (i) easing loop chain specification possibly through automatic detection, (ii) exploiting parallelism within the sparse tiles, (iii) optimizing the performance of the generalized full sparse tiling algorithm and investigating other ways to specialize it for each ap-

plication domain, and (iv) automating the process of tuning parameters to full sparse tiling.

Chapter 3

Cross-loop Optimization of Arithmetic Intensity for Finite Element Local Assembly

3.1 Introduction and Motivations

In many fields such as computational fluid dynamics, computational electromagnetics and structural mechanics, phenomena are modelled by partial differential equations (PDEs). Numerical techniques, like the finite volume method and the finite element method, are widely employed to approximate solutions of these PDEs. Unstructured meshes are often used to discretize the computational domain, since they allow an accurate representation of complex geometries. The solution is sought by applying suitable numerical operations, or kernels, to the entities of a 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 example, in Markall et al. [2013], Logg et al. [2012], AMCG [2010], DeVito et al. [2011].

The time required to apply the numerical kernels is a major issue, since the equation domain needs to be discretized into an extremely large number

of cells to obtain a satisfactory approximation of the PDE, possibly of the order of trillions, as in Rossinelli et al. [2013]. For example, it has been well established that mesh resolution is critical in the accuracy of numerical weather forecasts. However, operational forecast centers have a strict time limit in which to produce a forecast - 60 minutes in the case of the UK Met Office. Producing efficient kernels has a direct scientific payoff in higher resolution, and therefore more accurate, forecasts. Computational cost is a dominant problem in computational science simulations, especially for those based on finite elements, which are the subject of this work. In this chapter, we address, in particular, the well-known problem of optimizing the local assembly phase of the finite element method, which can be responsible for a significant fraction of the overall computation run-time, often in the range 30-60% [Russell and Kelly, 2013], [Olgaard and Wells, 2010], [Knepley and Terrel, 2013], [Kirby et al., 2005].

During the assembly phase, the solution of the PDE is approximated by executing a problem-specific kernel over all cells, or elements, in the discretized domain. We restrict our focus to relatively low order finite element methods, in which an assembly kernel's working set is usually small enough to fit the L1 cache. Low order methods are by no means exotic: they are employed in a wide variety of fields, including climate and ocean modeling, computational fluid dynamics, and structural mechanics. The efficient assembly of high order methods such as the spectral element method [Vos et al., 2010] requires a significantly different loop nest structure. High order methods are therefore excluded from our study.

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 the minimization of floating-point operations, register allocation and instruction-level parallelism, especially in the form of SIMD vectorization.

We aim to maximize our impact on the platforms that are realistically used for finite element applications, so we target conventional CPU architectures rather than GPUs. The key limiting factor to the execution on GPUs

is the stringent memory requirements. Only relatively small problems fit in a GPU memory, and support for distributed GPU execution in general purpose finite element frameworks is minimal. There has been some research on adapting local assembly to GPUs (mentioned later), although it differs from ours in several ways, including: (i) not relying on automated code generation from a domain-specific language (explained next), (ii) testing only very low order methods, (iii) not optimizing for cross-loop arithmetic intensity (the goal is rather effective multi-thread parallelization). In addition, our code transformations would drastically impact the GPU parallelization strategy, for example by increasing a thread's working set. For all these reasons, a study on extending the research to GPU architectures is beyond the scope of this work. In Section 3.10, however, we provide some intuitions about this research direction.

Achieving high-performance on CPUs is non-trivial. The complexity of the mathematical expressions, often characterized by a large number of operations on constants and small matrices, 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. We will show that traditional vendor compilers, such as *GNU's* and *Intel's*, fail at exploiting the structure inherent such assembly expressions. 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, as explained next.

We focus on optimizing the performance of local assembly operations produced by automated code generation. This technique has been proved successful in the context of the FEniCS [Logg et al., 2012] and Firedrake [Firedrake contributors, 2014] projects, become incredibly popular over the last years. In these frameworks, a mathematical model is expressed at high-level by means of a domain-specific language and a domain-specific compiler is used to produce a representation of local assembly operations (e.g. C code). *Our aim is to obtain close-to-peak performance in all of the local assembly operations that such frameworks can produce.* Since the domain-specific language exposed to the users provide as constructs generic differential operators, an incredibly vast set of PDEs, possibly arising in completely different domains, can be expressed and solved. A compiler-based approach is, there-

fore, the only reasonable option to the problem of optimizing local assembly operations.

Several studies have already tackled local assembly optimization in the context of automated code generation. In Olgaard and Wells [2010], it is shown how this technique can be leveraged to introduce domain-specific optimizations, which a user cannot be expected to write "by hand". Kirby et al. [2005] and Russell and Kelly [2013] have studied, instead, different optimization techniques based on a mathematical reformulation of the local assembly operations. The same problem has been addressed recently also for GPU architectures, for instance in Knepley and Terrel [2013], Klöckner et al. [2009], and Bana et al. [2014]. With our study, we make clear step forward by showing that different PDEs, on different platforms, require distinct sets of transformations if close-to-peak performance must be reached, and that low-level, domain-aware code transformations are essential to maximize instruction-level parallelism and register locality. As discussed in the following sections, our optimization strategy is quite different from those in previous work, although we reuse and leverage some of the ideas formulated in the available literature.

We present a novel structured approach to the optimization of automatically-generated local assembly kernels. We argue that for complex, realistic PDEs, peak performance can be achieved only by passing through a two-step optimization procedure: 1) expression rewriting, to minimize floating point operations, 2) and code specialization, to ensure effective register utilization and instruction-level parallelism, especially SIMD vectorization.

Expression rewriting consists of a framework capable of minimizing arithmetic intensity and optimize for register pressure. Our contribution is twofold:

- *Rewrite rules for assembly expressions.* The goal is to the reduce the computational intensity of local assembly kernels by rescheduling arithmetic operations based on a set of rewrite rules. These aggressively exploit associativity, distributivity, and commutativity of operators to expose loop-invariant sub-expressions and SIMD vectorization opportunities to the code specialization stage. While rewriting an assembly expression, domain knowledge is used in several ways, for example to avoid redundant computation.

- *An algorithm to deschedule useless operations.* Relying on symbolic execution, this algorithm restructures the code so as to skip useless arithmetic operations, for example multiplication by scalar quantities which are statically known to be zero. One problem is to transform the code while preserving code vectorizability, which is solved by resorting to domain-knowledge.

Code specialization's goal is to apply transformations to maximize the exploitation of the underlying platform's resources, e.g. SIMD lanes. We provide a number of contributions:

- *Padding and data alignment.* The small size of the loop nest (integration, test, and trial functions loops) require all of the involved arrays to be padded to a multiple of the vector register length so as to maximize the effectiveness of SIMD code. Data alignment can be enforced as a consequence of padding.

- *Vector-register Tiling.* Blocking at the level of vector registers, which we perform exploiting the specific memory access pattern of the assembly expressions (i.e. a domain-aware transformation), improves data locality beyond traditional unroll-and-jam optimizations. This is especially true for relatively high polynomial order (i.e. greater than 2) or when pre-multiplying functions are present.

- *Expression Splitting.* In certain assembly expressions the register pressure is significantly high: when the number of basis functions arrays (or, equivalently, temporaries introduced by loop-invariant code motion) and constants is large, spilling to L1 cache is a consequence for architectures with a relatively low number of logical registers (e.g. 16/32). We exploit sum's associativity to "split" the assembly expression into multiple sub-expressions, which are computed individually.

- *An algorithm to generate calls to BLAS routines.*

- *Autotuning.* We implement a model-driven, dynamic autotuner that transparently evaluates multiple sets of code transformations to determine the best optimization strategy for a given PDE. The main challenge here is to build, for a generic problem, a reasonably small search space that comprises most of the effective code variants.

Expression rewriting and code specialization have been implemented in a compiler, COFFEE[1], fully integrated with the Firedrake framework. Besides separating the mathematical domain, captured by a domain-specific compiler at an higher level of abstraction, from the optimization process, COFFEE also aims to be platform-agnostic. The code transformations occur on an intermediate representation of the assembly operation, which is ultimately translated into platform-specific code. Domain knowledge is exploited in two ways: for simplifying the implementation of code transformations and to make them extremely effective. Domain knowledge is conveyed to COFFEE from the higher level through suitable annotations attached to the input. For example, when the input is in the form of an abstract syntax tree produced by the higher layer, specific nodes are decorated so as to drive the optimization process. Although COFFEE has been thought of as a multi-platform optimizing compiler, our performance evaluation so far has been restricted to standard CPU platforms only. We emphasize once more, however, that all of the transformations applicable would work on generic accelerators as well.

To demonstrate the effectiveness of our approach, we provide an extensive and unprecedented performance evaluation across a number of real-world PDEs of increasing complexity, including some based on complex hyperelasticity models. We characterize our problems by varying polynomial order of the employed function spaces and number of so called pre-multiplying functions. To clearly distinguish the improvements achieved by COFFEE, we will compare, for each examined PDE, four sets of code variants: 1) unoptimized code, i.e. a local assembly routine as returned from the domain-specific compiler; 2) code optimized by FEniCS, i.e. the work in Olgaard and Wells [2010]; 3) code optimized by expression rewriting and code specialization as described in this paper. Notable performance improvements of 3) over 1) and 2) are reported and discussed.

3.2 Preliminaries

In this section, the basic concepts sustaining the finite element method are summarized. The notation adopted in Olgaard and Wells [2010] and Russell and Kelly [2013] is followed. At the end of this section, the reader is

---

[1]COFFEE stands for COmpiler For Finit Element local assEmbly.

expected to understand what local assembly represents and how an implementation can be derived starting from a mathematical specification of the finite element problem.

3.2.1 Overview of the Finite Element Method

We consider the weak formulation of a linear variational problem

$$
\begin{aligned}
&Find\ u\ \in U\ such\ that \\
&a(u, v) = L(v), \forall v \in V
\end{aligned}
\tag{3.1}
$$

where a and L are called bilinear and linear form, respectively. The set of *trial* functions U and the set of *test* functions V are discrete function spaces. For simplicity, we assume $U = V$ and $\{\phi_i\}$ be the set of basis functions spanning U. The unknown solution u can be approximated as a linear combination of the basis functions $\{\phi_i\}$. From the solution of the following linear system it is possible to determine a set of coefficients to express u

$$
A\mathbf{u} = b
\tag{3.2}
$$

in which A and b discretize a and L respectively:

$$
\begin{aligned}
A_{ij} &= a(\phi_i(x), \phi_j(x)) \\
b_i &= L(\phi_i(x))
\end{aligned}
\tag{3.3}
$$

The matrix A and the vector b are computed in the so called assembly phase. Then, in a subsequent phase, the linear system is solved, usually by means of an iterative method, and $\mathbf{u}$ is eventually evaluated.

We focus on the assembly phase, which is often characterized as a two-step procedure: *local* and *global* assembly. Optimizing the performance of local assembly is the subject of the research. Local assembly consists of computing the contributions that an element in the discretized domain provide to the approximated solution of the equation. Global assembly, on the other hand, is the process of suitably "inserting" such contributions in A and b.

3.2.2 Quadrature Representation for Finite Element Local Assembly

Without loss of generality, we illustrate local assembly in a concrete example, the evaluation of the local element matrix for a Laplacian operator. Consider the weighted Laplace equation

$$-\nabla \cdot (w\nabla u) = 0 \tag{3.4}$$

in which u is unknown, while w is prescribed. The bilinear form associated with the weak variational form of the equation is:

$$a(v, u) = \int_{\Omega} w\nabla v \cdot \nabla u \; \mathrm{d}x \tag{3.5}$$

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

$$A_{ij}^K = \int_K w\nabla\phi_i^K \cdot \nabla\phi_j^K \; \mathrm{d}x \tag{3.6}$$

The local element vector L can be determined in an analogous way starting from the linear form associated with the weak variational form of the equation.

Quadrature schemes are conveniently used to numerically evaluate A_{ij}^K. For convenience, a reference element K_0 and an affine mapping $F_K : K_0 \to K$ to any element $K \in T$ are introduced. This implies a change of variables from reference coordinates X_0 to real coordinates $x = F_K(X_0)$ is necessary any time a new element is evaluated. The numerical integration routine based on quadrature representation over an element K can be expressed as follows

$$A_{ij}^K = \sum_{q=1}^{N}\sum_{\alpha_3=1}^{n}\phi_{\alpha_3}(X^q)w_{\alpha_3}\sum_{\alpha_1=1}^{d}\sum_{\alpha_2=1}^{d}\sum_{\beta=1}^{d}\frac{\partial X_{\alpha_1}}{\partial x_\beta}\frac{\partial\phi_i^K(X^q)}{\partial X_{\alpha_1}}\frac{\partial X_{\alpha_2}}{\partial x_\beta}\frac{\partial\phi_j^K(X^q)}{\partial X_{\alpha_2}}detF_K'W^q \tag{3.7}$$

where N is the number of integration points, W^q the quadrature weight at the integration point X^q, d is the dimension of Ω, n the number of degrees of freedom associated to the local basis functions, and det the determinant of the Jacobian matrix used for the aforementioned change of coordinates.

In the next sections, we will often refer to the local element matrix evaluation, such as Equation 3.7 for the weighted Lapalce operator, as the *assembly expression* deriving from the weak variational problem.

3.2.3 Implementation of Quadrature-based Local Assembly

From Math to Code

We have explained that local assembly is the computation of contributions of a specific cell in the discretized domain to the linear system which yields the PDE solution. The process consists of numerically evaluating problem-specific integrals to produce a matrix and a vector (only the derivation of the matrix was shown in Section 3.2.2), whose sizes depend on the order of the method. This operation is applied to all cells in the discretized domain (mesh).

We consider again the weighted Laplace example of the previous section. A C-code implementation of Equation 3.7 is illustrated in Listing 4. The values at the various quadrature points of basis functions (ϕ) derivatives are tabulated in the A and B arrays. The summation along quadrature points q is implemented by the i loop, whereas the one along α_3 is represented by the r loop. In this example, we assume $d = 2$ (2D mesh), so the summations along α_1, α_2 and β have been straightforwardly expanded in the expression that evaluates the local element matrix A.

More complex assembly expressions, due to the employment of particular differential operators in the original PDE, are obviously possible. Intuitively, as the complexity of the PDE grows, the implementation of local assembly becomes increasingly more complicated. This fact is actually the real motivation behind reasearch in automated code generation techniques, such as those used by state-of-the-art frameworks like FEniCS and Firedrake. Automated code generation allows scientists to express the finite element specfication using a domain-specific language resembling mathematical notation, and obtain with no effort a semantically correct implementation of local assembly. The research in the present work is about making such an implementation also extremely effective, in terms of run-time performance, on standard CPU architectures.

The domain-specific language used by Firedrake and FEniCS to express finite element problems is the Unified Form Language (UFL) [Alnæs et al.,

LISTING 4: A possible implementation of Equation 3.7 assuming a 2D triangular mesh and polynomial order $q = 2$ Lagrange basis functions.

```
 1  void weighted_laplace(double A[3][3], double **coords, double **w)
 2  {
 3    // Compute Jacobian
 4    double J[4];
 5    compute_jacobian_triangle_2d(J, coords);
 6
 7    // Compute Jacobian inverse and determinant
 8    double K[4];
 9    double detJ;
10    compute_jacobian_inverse_triangle_2d(K, detJ, J);
11    const double det = fabs(detJ);
12
13    // Quadrature weights
14    static const double W1[1] = 0.5;
15
16    // Basis functions
17    static const double A[1][3] = {{ -0.999999999999999, ...}} ;
18    static const double B[1][3] = {{ -1.0, ...}} ;
19    static const double C[1][3] = {{ 0.333333333333333, ...}} ;
20
21    for (int i = 0; i ¡ 6; ++i)
22    {
23      double f0 = 0.0;
24      for (int r = 0; r ¡ 3; ++r)
25        f0 += (w[r][0] * C[i][r]);
26
27      for (int j = 0; j ¡ 3; ++j)
28        for (int k = 0; k ¡ 3; ++k)
29          A[j][k] += (((((K[1]*A[i][k])+(K[3]*B[i][k])) *
30                        ((K[1]*A[i][j])+(K[3]*B[i][j]))) +
31                        (((K[0]*A[i][k])+(K[2]*B[i][k])) *
32                        ((K[0]*A[i][j])+(K[2]*B[i][j]))))*det*W1[i]*f0);
33    }
34  }
```

2014]. Listing 5 shows a possible UFL implementation for the weighted Laplace form. Note the resemblance of $a = weight^*...$ with Equation 3.6. A form compiler translates UFL code into the C code shown in Listing 4. We will describe these aspects carefully in Section 3.7; for the moment, this level of detail suffices to open a discussion on how to optimize local assembly kernels arising from generic partial differential equations.

Other Examples and Motivations for Optimizations

The structure of a local assembly kernel can be generalized as in Figure 3.1. The inputs are a zero-initialized two dimensional array used to store the element matrix, the element's coordinates in the discretized domain, and coefficient fields, for instance indicating the values of velocity or pressure in

40

LISTING 5: UFL specification of the weighted Laplace equation for polynomial order $q = 2$ Lagrange basis functions.

```
1  // This is a Firedrake construct (not an UFL's) to instantiate a 2D mesh.
2  mesh = UnitSquareMesh(size, size)
3  // FunctionSpace also belongs to the Firedrake language
4  V = FunctionSpace(mesh, "Lagrange", 2)
5  u = TrialFunction(V)
6  v = TestFunction(V)
7  weight = Function(V).assign(value)
8  a = weight*dot(grad(v), grad(u))*dx
```

```
Input: element matrix (2D array, initialized to 0), coordinates (array),
       coefficients (array, e.g. velocity)
Output: element matrix (2D array)
- Compute Jacobian from coordinates
- Define basis functions
- Compute element matrix in an affine loop nest
```

Figure 3.1: Structure of a local assembly kernel

the element. The output is the evaluated element matrix. The kernel body can be logically split into three parts:

1. Calculation of the Jacobian matrix, its determinant and its inverse required for the aforementioned change of coordinates from the reference element to the one being computed.

2. Definition of basis functions used to interpolate fields at the quadrature points in the element. The choice of basis functions is expressed in UFL directly by users. In the generated code, they are represented as global read-only two dimensional arrays (i.e., using `static const` in C) of double precision floats.

3. Evaluation of the element matrix in an affine loop nest, in which the integration is performed.

Table 3.1 shows the variable names we will use in the upcoming code snippets to refer to the various kernel objects.

The actual complexity of a local assembly kernel depends on the finite element problem being solved. In simpler cases, the loop nest is perfect, has short trip counts (in the range 3–15), and the computation reduces to a summation of a few products involving basis functions. An example is provided in Listing 6, which shows the assembly kernel for a Helmholtz

Object name	Type	Variable name(s)
Determinant of the Jacobian matrix	double	det
Inverse of the Jacobian matrix	double	K1, K2, ...
Coordinates	double**	coords
Fields (coefficients)	double**	w
Coefficients at quadrature points	double	f0, f1, ...
Numerical integration weights	double[]	W
Basis functions (and derivatives)	double[][]	A, B, C, ...
Element matrix	double[][]	M

Table 3.1: Type and variable names used in the various listings to identify local assembly objects.

LISTING 6: Local assembly implementation for a Helmholtz problem on a 2D mesh using polynomial order $q = 1$ Lagrange basis functions.

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

problem using Lagrange basis functions on 2D elements with polynomial order $q = 1$. In other scenarios, for instance when solving the Burgers equation, the number of arrays involved in the computation of the element matrix can be much larger. The assembly code is given in Listing 7 and contains 14 unique arrays that are accessed, where the same array can be referenced multiple times within the same expression. This may also require the evaluation of constants in outer loops (called F in the code) to act as scaling factors of arrays. Trip counts grow proportionally to the order of the method and arrays may be block-sparse.

In general, the variations in the structure of mathematical expressions and in loop trip counts (although typically limited to the order of tens of iterations) that different equations show, render the optimization process challenging, requiring distinct sets of transformations to bring performance closest to the machine peak. For example, the Burgers problem, given the

LISTING 7: Local assembly implementation for a Burgers problem on a 3D mesh using polynomial order $q = 1$ Lagrange basis functions.

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

large number of arrays accessed, suffers from high register pressure, whereas the Helmholtz equation does not. Moreover, arrays in Burgers are block-sparse due to the use of vector-valued basis functions (we will elaborate on this in the next sections). These few aspects (we could actually find more) already intuitively suggests that the two problems require a different treatment, based on an in-depth analysis of both data and iteration spaces. Furthermore, domain knowledge enables transformations that a general-purpose compiler could not apply, making the optimization space even larger. In this context, our goal is to understand the relationship between distinct code transformations, their impact on cross-loop arithmetic intensity, and to what extent their composability is effective in a wide class of real-world equations and architectures.

We also note that despite the infinite variety of assembly kernels that frameworks like FEniCS and Firedrake can generate, it is still possible to identify common domain-specific traits that are potentially exploitable for our optimization strategy. These include: 1) memory accesses along the three loop dimensions are always unit stride; 2) the **j** and **k** loops

are interchangeable, whereas interchanges involving the i loop require pre-computation of values (e.g. the F values in Burgers) and introduction of temporary arrays (explained next); 3) depending on the problem being solved, the `j` and `k` loops could iterate along the same iteration space; 4) most of the sub-expressions on the right hand side of the element matrix computation depend on just two loops (either `i-j` or `i-k`). In the following sections we show how to exploit these observations to define a set of systematic, composable optimizations.

3.3 Overview of the Optimization Strategy

To generate high performance implementation of local assembly kernels, assembly expressions must be optimized with regards to three interrelated aspects:

- arithmetic intensity

- instruction-level parallelism

- data locality

Three conceptually distinct kind of transformations can be individuated, which we refer to as *expression rewriting*, *code specialization*, and *general-purpose optimizations*. Expression rewriting mainly targets arithmetic intensity by transforming the assembly expression (and its enclosing loop nest) so as to minimize the number of floating point operations required to evaluate the local element matrix. Code specialization is tailored to optimizing for instruction-level parallelism, particularly SIMD vectorization, and data (register) locality. Both classes of transformations are inspired by the inherent structure of local assembly code and make use of domain knowledge: as elaborated in the next sections, these two aspects are the core motivations for which standard general-purpose compilers fail at maximizing the performance of local assembly kernels. The third class of transformations is about general-purpose optimizations; that is, generic, well-known techniques for improving code performance, such as loop unrolling or loop interchange, that for some reasons are not applied by the compiler generating machine code, but potentially useful in certain equations.

In Sections 3.4–3.6, these classes of code transformations are presented. In Section 3.7, it is explained that their application to local assembly kernels must follow a specific order to ensure the correctness of the resulting code. Some effort was invested in ensuring that optimizations at stage i (e.g. expression rewriting) would not break any further optimization opportunity at stage $i+1$ (code specialization). The theory and the implementation of a technique to select the optimal combination of transformations for a given equation are finally discussed.

3.4 Expression Rewriting

Expression rewriting is about exploiting properties of operators such as associativity, distributivity, and commutativity, to minimize arithmetic intensity, expose code vectorization opportunities, and optimize the register pressure in the various levels of the assembly loop nest. There are many possibilities of rewriting an expression, so the transformation space can be quite large. Firstly, in Sections 3.4.1–3.4.4, several ways of manipulating an assembly expression and their potential impact on the computational cost are described. Then, in Section 3.4.5, a simple yet systematic way of rewriting an expression based on a set of formal rewrite rules, which is the key to effectively explore the expression's transformation space, is formalized.

The second objective of expression rewriting (Section 3.4.6) consists of restructuring the iteration space so as to avoid arithmetic operations over zero-valued columns in block-sparse basis functions arrays. Zero-valued columns arise, for example, when taking derivatives on a reference element or when using mixed (vector-valued) elements. This problem was tackled in Olgaard and Wells [2010], but the proposed solution, which makes use of indirection arrays in the generated local assembly code, breaks most of the optimizations applicable by code specialization, including the fundamental SIMD vectorization. The contribution of the present research is a novel domain-aware approach based on symbolic execution that avoids indirection arrays.

3.4.1 Generalized Loop-invariant Code Motion

Consider the local element matrix computation in Figure **??**, which is an excerpt from the Burgers problem shown in Listing 7. The assembly ex-

```
for (int i=0; i<I; ++i) {
  // Coefficient evaluation at point i
  for (int r=0; r<T; ++r) {
    f0 += ...; f1 += ...; ...;
  }
  // Test functions
  for (int j=0; j<T; ++j)
  {
    // Trial functions
    for (int k=0; k<T; ++k)
    {
      M[j][k] += (((A[i][k]/c)*A[i][j])+
                  ((a*f0*A[i][j]+b*f1*B[i][j])*A[i][k]*g)+
                (((d*D[i][k]+e*E[i][k])*A[i][j])*f))*det*W[i]
    }
  }
}
```

Figure 3.2: Original (simplified) code

pression, produced by the FEniCS and Firedrake's form compiler, has been deliberately simplified, and code details have been omitted for brevity and readability. In practice, as already emphasized, assembly expressions can be much more complex depending on the differential operators employed in the variational form; however, this example is representative enough for highlighting patterns that are common in a large class of equations.

A first glimpse of the code suggests that the `a*f0*A[i][j]+b*f1*B[i][j]` sub-expression is invariant with respect to the innermost (trial functions) loop `k`, so it can be hoisted at the level of the outer loop `j` to avoid redundant computation. This is indeed a standard compiler transformation, supported by any available compilers, so, in principle, there should be no need to transform the source code explicitly. With a closer look we notice that the sub-expression `d*D[i][k]+e*E[i][k]` is also invariant, although, this time, with respect to the outer (test functions) loop `j`. Available compilers (e.g. *GNU's* and *Intel's*) limit the search for code motion opportunities to the innermost loop of a given nest. Moreover, the hoisted code is scalar and therefore not subjected to SIMD auto-vectorization. In other words, these general-purpose compilers lack performance models to determine (i) the optimal place where to hoist an expression and (ii) the potential gain and

```
for ( int i=0; i<I ; ++i ) {
  for ( int r=0; r<T; ++r ) {
    f0 += ...; f1 += ...; ...;
  }
  double T1[T] , T2[T];
  for ( int r=0; r<T; ++r ) {
    T1[ r ] = (( f0*a*A[ i ][ r ])+(f1*b*B[ i ][ r ]));
    T2[ r ] = ((d*D[ i ][ r ])+(e*E[ i ][ r ]));
  }
  for ( int j=0; j<T; ++j ) {
    for ( int k=0; k<T; ++k ) {
      M[ j ][ k ] += (((A[ i ][ k ]/ c )*A[ i ][ j ])+
                      (T1[ j ]*A[ i ][ k ]*g)+
        (T2[ k ]*A[ i ][ j ]* f ))*det*W[ i ];
    }
  }
}
```

Figure 3.3: Invariant code

overhead (due to the need for extra temporary memory) of vectorization. These are notable limitations for local assembly kernels.

We work around these limitations with source-level loop-invariant code motion. In particular, we pre-compute all values that an invariant sub-expression assumes along its fastest varying dimension. This is implemented by introducing a temporary array per invariant sub-expression and by adding a new loop to the nest. At the price of extra memory for storing temporaries, the gain is that lifted terms can be auto-vectorized as part of an inner loop. Given the short trip counts of our loops, it is important to achieve auto-vectorization of hoisted terms in order to minimize the percentage of scalar instructions, which could otherwise be significant. It is also worth noting that, in some problems, for instance Helmholtz in Listing 6, invariant sub-expressions along **j** are identical to those along **k**, and both loops iterate over the same iteration space, as anticipated in Section 3.2.3. In these cases, we safely avoid redundant pre-computation. The resulting code for the running Burgers example is shown in Figure **??**.

In the following, we refer to this series of transformations as *generalized loop-invariant code motion*. We will show that this optimization is crucial when optimizing non-trivial assembly expressions, allowing to achieve

```
for (int i=0; i<I; ++i) {
  for (int r=0; r<T; ++r) {
    f0 += ...; f1 += ...; ...;
  }
  double T1[T], T2[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((f0*a*A[i][r])+(f1*b*B[i][r]));
    T2[r] = ((d*D[i][r])+(e*E[i][r]));
    T3[r] = ((A[i][r]/c)+T2[r]*f);
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += ((T1[j]*A[i][k]*g)+
                  (T3[k]*A[i][j]))*det*W[i];
    }
  }
}
```

Figure 3.4: Factorized code

performance improvements over the original code larger than $3\times$.

3.4.2 Terms Factorization

After generalized loop-invariant code motion has been applied, some assembly expressions can still "hide" opportunities for code hoisting. By examining again the code in Figure **??**, we notice that the basis function array `A` iterating along the `[i,j]` loops appears twice in the expression. By expanding the products in which `A` is accessed and by applying sum commutativity, terms can be factorized. This has two effects: firstly, it reduces the number of arithmetic operations performed; secondly, and most importantly, it exposes a new sub-expression `A[i][k]/c+T2[k]*f` invariant with respect to loop `j`. Consequently, hoisting can be performed, resulting in the code in Figure **??**. In general, exposing factorization opportunities requires traversing the whole expression tree, and then expanding and moving terms. It also needs heuristics to select a factorization strategy: there may be different opportunities of reorganizing sub-expressions, and, in our case, the best is the one that maximizes the invariant code eventually disclosed. We will discuss this aspect formally in Section 3.4.5.

```
for (int i=0; i<I; ++i) {
  for (int r=0; r<T; ++r) {
    f0[i] += ...; f1[i] += ...; ...;
  }
  k0[i] = f0[i]*a; k1[i] = f1*b;
}
for (int i=0; i<I; ++i) {
  double T1[T], T2[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((k0[i]*A[i][r])+(k1[i]*B[i][r]));
    T2[r] = ((d*D[i][r])+(e*E[i][r]));
    T3[r] = ((A[i][r]/c)+T2[r]*f);
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += ((T1[j]*A[i][k]*g)+
                  (T3[k]*A[i][j]))*det*W[i];
    }
  }
}
```

Figure 3.5: Scalar-expanded code

3.4.3 Precomputation of Invariant Terms

We note that integration-dependent expressions are inherently executed as scalar code. For example, the `f0*a` and `f1*b` products in Figure **??** depend on the loop along quadrature points; these operations are performed in a non-vectorized way at every `i` iteration. This is not a big issue in our running example, in which the scalar computation represents a small fraction of the total, but it becomes a concrete problem in complicated forms, like those at the heart of hyperelasticity (which will be part of our performance evaluation). In such forms, the amount of computation independent of both test and trial functions loops is so large that it has a significant impact on the run-time, despite being executed only $O(I)$ times (with I number of quadrature points). We have therefore implemented an algorithm to move and scalar-expand integration-dependent expressions, which leads to codes as in Figure **??**.

3.4.4 Expanding Sub-expressions

Expression rewriting also aims at minimizing register pressure in the assembly loop nest. Once the code has been optimized for arithmetic intensity,

it is important to think about how the transformations impacted register allocation. Assume the local assembly kernel is executed on a state-of-the-art CPU architecture having 16 logical registers, e.g. an Intel Haswell. Each value appearing in the expression is loaded and kept in a register as long as possible. In Figure **??**, for instance, the scalar value `g` is loaded once, whereas the term `det*W[i]` is precomputed and loaded in a register at every `i` iteration. This implies that at every iteration of the `[j,k]` loop nest, 12% of the available registers are spent just to store values independent of test and trial functions loops. In more complicated expressions, the percentage of registers destined to store such constant terms can be even higher. Registers are, however, a precious resource, especially when evaluating compute-intensive expressions. The smaller is the number of available free registers, the worse is the instruction-level parallelism achieved: for example, a shortage of registers can increase the pressure on the L1 cache (i.e. it can worsen data locality), or it may prevent the effective application of standard transformations, e.g. loop unrolling. We aim at relieving this problem by suitably expanding terms and introducing, where necessary, additional temporary values. We illustrate this in the following example.

Consider a variant of the Burgers local assembly kernel, shown in Figure 3.6(a). This is again a representative, simplified example. We can easily distribute `det*W[i]` over the three operands on the left-hand side of the multiplication, and then absorb it in the pre-computation of the invariant sub-expression stored in `T1`, resulting in code as in Figure 3.6(b). Freeing the register destined to the constant `g` is less straightforward: we cannot absorb it in `T1` as we did with `det*W[i]` since `T1` is also accessed in the `T1[j]*A[i][k]` sub-expression. The solution is to add another temporary as in Figure 3.6(c). Generalizing, this is a problem of data dependencies: to solve it, we use a dependency graph in which we add a direct edge from identifier `A` to identifier `B` to denote that the evaluation of `B` depends on `A`. The dependency graph is initially empty, and is updated every time a new temporary is created by either loop-invariant code motion or expansion of terms. The dependency graph is then queried to understand when expansion can be performed without resorting to new temporary values. This aspect is formalized in the next section.

50

```
for (int i=0; i<I; ++i) {
  double T1[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((a*A[i][r])+(b*B[i][r]));
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += (((T1[j]*A[i][k])+(T1[k]*A[i][j]))*g+
                  +(T1[j]*A[i][k]))*det*W[i];
    }
  }
}
```

(a) Original (simplified) code

```
for (int i=0; i<I; ++i) {
  double T1[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((a*A[i][r])+(b*B[i][r]))*det*W[i];
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += (T1[j]*A[i][k]+T1[k]*A[i][j])*g+(T1[j]*A[i][k]);
    }
  }
}
```

(b) The code after expansion of `det*W[i]`

```
for (int i=0; i<I; ++i) {
  double T1[T], T2[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((a*A[i][r])+(b*B[i][r]))*det*W[i];
    T2[r] = T1[r]*g;
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += (T2[j]*A[i][k])+(T2[k]*A[i][j])+(T1[j]*A[i][k]);
    }
  }
}
```
(c) The code after expansion of `g`. Note the need to introduce a new temporary array.

Figure 3.6: Expansion of terms to improve register pressure in a local assembly kernel

3.4.5 Rewrite Rules for Assembly Expressions

In general, assembly expressions produced by automated code generation can be much more complex than those we have used as examples, with dozens of terms involved (basis function arrays, derivatives, coefficients, ...) and hundreds of (nested) arithmetic operations. Our goal is to establish a portable, unified, platform-independent, and systematic way of reducing the computational strength of an expression exploiting the intuitions described in the previous section. This *expression rewriting* should be simple; definitely, it must be robust to be integrated in an optimizing domain-specific compiler capable of supporting real problems, such as COFFEE. In other words, we look for an algorithm capable of transforming a plain assembly expression by applying 1) generalized loop-invariant code motion, 2) non-trivial factorization and re-association of sub-expressions, and 3) expansion of terms; after that, it should perform 4) scalar-expansion of integration-dependent terms to achieve full vectorization of the assembly code.

We centre such an algorithm around a set of rewrite rules. These rules drive the transformation of an expression, prescribe where invariant sub-expressions will be moved (i.e. at what level in the loop nest), and track the propagation of data dependencies. When applying a rule, the state of the loop nest must be updated to reflect, for example, the use of a new temporary and new data dependencies. We define the state of a loop nest as $L = (\sigma, G)$, where $G = (V, E)$ represents the dependency graph, while $\sigma : Inv \rightarrow S$ maps invariant sub-expressions to identifiers of temporary arrays. The notation σ_i refers to invariants hoisted at the level of loop i. We also introduce the *conditional hoister* operator $[]$ on σ such that

$$\sigma[v/x] = \begin{cases} \sigma(x) & \text{if } x \in Inv; \ v \text{ is ignored} \\ v & \text{if } x \notin Inv; \ \sigma(\text{x}) = \text{v} \end{cases}$$

That is, if the invariant expression **x** has already been hoisted, $[]$ returns the temporary identifier hosting its value; otherwise, **x** is hoisted and a new temporary v is created. There is a special case when $v = \perp$, used for

Rule	Precondition

$$[a_i \cdot b_j]_{(\sigma, G)} \rightarrow [a_i \cdot b_j]_{(\sigma, G)}$$

$$[(a_i + b_j) \cdot \alpha]_{(\sigma, G)} \rightarrow [(a_i \cdot \alpha + b_j \cdot \alpha)]_{(\sigma, G)}$$

$$[a_i \cdot b_j + a_i \cdot c_j]_{(\sigma, G)} \rightarrow [(a_i \cdot (b_j + c_j))]_{(\sigma, G)}$$

$$[a_i + b_i]_{(\sigma, G)} \rightarrow [t_i]_{(\sigma', G')} \qquad t_i = \sigma_{i_0}[t'_i/a_i + b_i], G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, b_i)\})$$

$$[(a_i \cdot b_j) \cdot \alpha]_{(\sigma, G)} \rightarrow [t_i \cdot b_j]_{(\sigma', G')} \qquad \sharp(b_j) > \sharp(a_i), t_i = \sigma_{i_0}[\sigma[\perp/a_i]/a_i \cdot \alpha], a_i \notin in(G),$$

$$G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, \alpha)\})$$

$$[(a_i \cdot b_j) \cdot \alpha]_{(\sigma, G)} \rightarrow [t_i \cdot b_j]_{(\sigma', G')} \qquad \sharp(b_j) > \sharp(a_i), t_i = \sigma_{i_0}[t'_i/a_i \cdot \alpha], a_i \in in(G),$$

$$G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, \alpha)\})$$

Figure 3.7: Rewrite rules driving the rewriting process of an assembly expression.

conditional deletion of entries in σ. Specifically

$$\sigma[\perp/x] = \begin{cases} \sigma(x) & \text{if } x \in Inv; \ \sigma = \sigma \setminus (x, \sigma(x)) \\ v & \text{if } x \notin Inv; \ v \notin S \end{cases}$$

In other words, the invariant sub-expression **x** is removed and the temporary identifier that was hosting its value is returned if **x** had been previously hoisted; otherwise, a fresh identifier v is returned. This is useful to express updates of hoisted invariant sub-expressions when expanding terms.

In the following, a generic (sub-)expression is represented with Roman letters **a, b, ...**; constant terms are considered a special case, so Greek letters α, β, **...** are used instead. The iteration vector $i = [i_0, i_1, ...]$ is the ordered sequence of the indices of the loops enclosing an (sub-)expression. We will refer to i_0 as the outermost enclosing loop. The notation a_i, therefore, indicates that the expression a assumes distinct values while iterating along the loops in i; its outermost loop is silently assumed to be i_0.

Rewrite rules for expression rewriting are provided in Figure 3.7; obvious rules are omitted for brevity. The Expression Rewriter applies the rules while performing a depth-first traversal of the assembly expression tree. Given an arithmetic operation between two sub-expressions (i.e. a node in the expression tree), we first need to find an applicable rule. There cannot

be ambiguities: only one rule can be matched. If the preconditions of the rule are satisfied, the corresponding transformation is performed; otherwise, no rewriting is performed, and the traversal proceeds. As examples, it is possible to instantiate the rules in the codes shown in Figures **??** and 3.6(a); eventually, the optimized codes in Figures **??** and 3.6(c) are obtained, respectively.

To what extent should rewrite rules be applied is a question that cannot be answered in general. In some problems, a full rewrite of the expression may be the best option; in other cases, on the other hand, an aggressive expansion of terms, for example, may lead to high register pressure in the loops computing invariant terms, worsening the performance. In Section 3.5 how to leverage code specialization to select a suitable rewriting strategy for the problem at hand is explained.

3.4.6 Avoiding Iteration over Zero-valued Blocks by Symbolic Execution

Skipping arithmetic operations over blocks of zero-valued entries in basis functions arrays is the second goal of expression rewriting. Zero-valued columns arise, for example, when taking derivatives on a reference element and when employing vector-valued elements. In Olgaard and Wells [2010], a technique to avoid operations on zero-valued columns based on the use of indirection arrays (e.g. `A[B[i]]`, where `A` is a tabulated basis function and `B` a map from loop iterations to non-zero columns in `A`) was described and implemented in FEniCS. The approach proposed in this section will be evaluated and compared to this pioneering work. Essentially, our strategy avoids indirection arrays in the generated code, which otherwise would break the optimizations applicable by code specialization, including SIMD vectorization.

In Figure 3.8(a), an enriched version of the Burgers excerpt in Figure **??** is illustrated. The code is instantiated for the specific case of polynomial order $q = 1$ Lagrange basis functions on a 2D mesh. The array `D` represents a derivative of a basis function tabulated at the various quadrature points. There are four zero-valued columns. Any multiplications or additions along these columns could (should) be skipped to avoid irrelevant floating point operations. The solution adopted in Olgaard and Wells [2010] is not to

54

```
void burgers(double M[6][6],
             double **coordinates,
             double **coeff0) {
   // Calculate determinant of jacobian
   // (det) using the coordinates field

   // Define tabulation of basis functions
   // (and their derivatives) arrays
   ...
   @coffee mfs[3, 5]
   static const double D[6][6] =
      {{0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
       {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
       {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
       {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
       {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
       {0.0, 0.0, 0.0, -1.0, 0.0, 1.0}};
   ...
   for (int i=0; i<6; ++i) {
      ...
      double T1[6], T2[6];
      for (int r=0; r<6; ++r) {
         T1[r] = a*A[i][r]+b*B[i][r];
         T2[r] = d*D[i][k]+e*E[i][k];
      }
      for (int j=0; j<6; ++j)
         for (int k=0; k<6; ++k)
            M[j][k] += (T1[j], T2[k], A[i][j], ...)
   }
}
```

(a) Original code. Note the annotation over the definition of the tabulated basis function D, which is used to identify the presence of zero-valued columns

```
void burgers(double M[6][6],
             double **coordinates,
             double **coeff0) {
   // Calculate determinant of jacobian (det)
   // using the coordinates field

   // Define tabulation of basis functions
   // (and their derivatives) arrays
   ...
   @coffee mfs[3, 5]
   static const double D[6][6] =
      {{0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
       {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
       {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
       {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
       {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
       {0.0, 0.0, 0.0, -1.0, 0.0, 1.0}};
   ...
   for (int i=0; i<6; ++i) {
      ...
      double T1[6], T2[6];
      for (int r=0; r<6; ++r) {
         T1[r] = a*A[i][r]+b*B[i][r];
         T2[r] = d*D[i][k]+e*E[i][k];
      }
      for (int j0=0; j0<3; ++j0)
         for (int k0=0; k0<3; ++k0)
            M[j0+3][k0+3] += f(T1[j0+3], A[i][k0+3], ...);
      for (int j1=0; j1<3; ++j1)
         for (int k1=0; k1<6; ++k1)
            M[j1][k1] += g(T2[k1], A[i][j1+3], ...);
   }
}
```

(b) The code after symbolic execution took place

Figure 3.8: Simplified excerpt of local assembly code from a Burgers form using vector-valued basis functions, before and after symbolic execution is performed to rewrite the iteration space

generate the zero-valued columns (i.e. to generate a dense 6×2 array), to reduce the size of the iteration space over test and trial functions (from 6 to 2), and to use an indirection array (e.g. $ind = \{3,5\}$) to update the right entries in the element tensor A. This prevents, among the various optimizations, effective SIMD vectorization, because memory loads and store would eventually reference non-contiguous locations.

Our new strategy exploits domain knowledge and makes use of symbolic execution. We discern the origin of zero-valued columns: for example, those due to taking derivatives on the reference element from those inherent to using vector-valued elements. In the running Burgers example, the use of vector-valued basis functions required the introduction of a zero-valued block (columns 0, 1, 2 in the array D) to correctly evaluate the local element matrix while iterating along the space of test and trial functions. The two key observations are that: (i) the number of zero-valued columns caused by using vector function spaces is, often, much larger then that due to

derivatives, and (ii) such columns are contiguous in memory. Based on this observation, we aim to avoid iteration only along the block of zero-valued columns induced by vector-valued elements.

The goal is achieved by means of symbolic execution. The algorithm expects some indication about the location of the zero-valued columns induced by vector-valued function spaces, for each tabulated basis function, as shown in Figure 3.8(a). Then, after expression rewriting took place, each statement is executed symbolically. For example, consider the assignment `T2[r] = d*D[i][k]+e*E[i][k]` in Figure 3.8(a). Array `D` has non-zero-valued columns in the range $NZ_D = [3, 5]$, while array `E` has non-zero-valued columns in the range $NZ_E = [0, 2]$, although not displayed. Multiplications by scalar quantities (e.g. `d*D[i][k]`) do not affect the propagation of non-zero-valued columns. On the other hand, summing two operands such as `d*D[i][k]` and `e*E[i][k]` requires tracking the fact that the target identifier `T2` will have non-zero-valued columns in the range $NZ_E \| NZ_D = [0, 5]$. Eventually, exploiting the NZ information computed and associated with each identifier, we split the original assembly expression into multiple sets of sub-expressions, each set characterized by the same range of non-zero-valued columns. In our example, assuming that $NZ_{T1} = [3, 5]$ and $NZ_A = [3, 5]$, there are two of such sets, which leads to the generation of two distinct iteration spaces (one for each set), as in Figure 3.8(b).

3.5 Code Specialization

Code specialization's goal is architecture-specific optimization for instruction-level parallelism and register locality. A number of transformations are provided at this stage, including those enabling effective SIMD vectorization.

3.5.1 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 are allocated

to addresses that are multiples of `VL`. Then, two dimensional arrays are padded by rounding the number of columns 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. 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.

Padding of all two dimensional arrays involved in the evaluation of the element matrix 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.

Example

Consider again the code in Figure 3.8(b) and assume `VL` = 4 double-precision floats (i.e. vector registers are 32 bytes longs).

The arrays in the loop nest `[j1,k1]` can be padded and the right bound of loop `k1` can be safely increased to 8: eventually, values computed in the region `M[0:3][6:8]` will be discarded. Then, by explicitly aligning arrays using suitable qualifiers (e.g. #`__attribute__((aligned(32)))` for the Intel compiler), effective SIMD auto-vectorization can be obtained for this loop nest.

There are some complications in the case of loops `[j0,k0]`. Here, increasing the innermost loop bound to 4 is still safe assuming that both `T1` and `A` are padded, but it has no effect: the starting addresses of the load instructions would be `&T1[3]` and `&A[i][3]`, which are clearly not aligned. Changing the starting address of `A` and `T1` is, in general, not an option, because these arrays could be accesses also in other loop nests, as happens with `A` in this example. One solution, instead, is to start iterating from the closest index that would ensure data alignment; in this case, $k0 = 0$. How-

ever, this would imply losing the effect of the zero-avoidance transformation (partially in general, totally for this loop nest). Another possibility is to attain to non-aligned accesses. Which of the two strategies is better cannot be established a-priori. An autotuning system, as described in Section 3.7.4, will help answering this question.

Finally, note that whenever increasing a loop bound cause accessing non-zero entries in the local element matrix M, there is no way of recovering data alignment.

3.5.2 Expression Splitting

In complex kernels, like Burgers in Listing 7, and on certain architectures, achieving effective register allocation can be challenging. If the number of variables independent of the innermost-loop dimension is close to or greater than the number of available CPU registers, poor register reuse is likely. This usually happens when the number of basis function arrays, temporaries introduced by generalized loop-invariant code motion, and problem constants is large. For example, applying loop-invariant code motion to Burgers on a 3D mesh requires 24 temporaries for the ijk loop order. This can make hoisting of the invariant loads out of the k loop inefficient on architectures with a relatively low number of registers. One potential solution to this problem consists of suitably "splitting" the computation of the element matrix A into multiple sub-expressions; an example, for the Burgers problem, is given in Listing **??**. The transformation can be regarded as a special case of classic loop fission, in which associativity of the sum is exploited to distribute the expression across multiple loops. To the best of our knowledge, expression splitting is not supported by available compilers.

Splitting an expression (henceforth *split*) has, however, several drawbacks. Firstly, it increases the number of accesses to A proportionally to the "split factor", which is the number of sub-expressions produced. Also, depending on how the split is executed and the way expression rewriting is performed, it can lead to redundant computation. For example, the number of times the product $det * W3[i]$ is performed is proportional to the number of sub-expressions, as shown in the code snippet. Further, it increases loop overhead, for example through additional branch instructions. Finally, it might affect register locality: for instance, the same array could be accessed

```
for (int i=0; i<I; ++i) {
  double T1[T], T2[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((a*A[i][r])+(b*B[i][r]))*det*W[i];
    T2[r] = T1[r]*g;
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += (T2[j]*A[i][k])+(T2[k]*A[i][j]);
    }
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += (T1[j]*A[i][k]);
    }
  }
}
```

Figure 3.9: Local assembly code for the Burgers problem in Figure 3.6(c) after the application of *split*. In this example, the split factor is 2.

in different sub-expressions, requiring a proportional number of loads be performed; this, again, depends on expression rewriting. Nevertheless, as shown in Section 3.8.2, the performance gain from improved register reuse along inner dimensions can be important, especially if the selection of the split factor uses heuristics to minimize the aforementioned issues.

3.5.3 Model-driven Vector-register Tiling

One notable problem of assembly kernels concerns register allocation and register locality. The critical situation occurs when loop trip counts and the variables accessed are such that the vector-register pressure is high. Since the kernel's working set fits the L1 cache, it is particularly important to optimize register management. Standard optimizations, such as loop interchange, unroll, and unroll-and-jam, can be employed to deal with this problem. Tiling at the level of vector registers represents another opportunity. Based on the observation that the evaluation of the element matrix can be reduced to a summation of outer products along the **j** and **k** dimensions, a model-driven vector-register tiling strategy can be implemented. If we consider the codes in the various listings and we focus on the body of the test and trial functions loops (**j** and **k**), the computation of the element

```
void burgers(double M[6][6], double **coordinates) {
  // Calculate determinant of jacobian (det) using the coordinates
  //    field
  // Define tabulation of basis functions (and their derivatives)
  //    arrays
  ...

  for (int i=0; i<6; ++i) {
    ...
    double T1[6], T2[6];
    for (int r=0; r<6; ++r) {
      T1[r] = a*A[i][r]+b*B[i][r];
      T2[r] = d*D[i][k]+e*E[i][k];
    }

    for (int j=0; j<4; ++j) {
      for (int k=0; k<8; ++k) {
        // "load" and "set" intrinsics
        // Compute M[0,0], M[1,1], M[2,2], M[3,3]
        // One "permute_pd" intrinsic per k-loop "load"
        // Compute M[0,1], M[1,0], M[2,3], M[3,2]
        // One "permute2f128_pd" intrinsic per k-loop load
        ...
      }
    }

    // Reminder loop
    for (int j=4; j<6; ++j) {
      for (int k=0; k<6; ++k) {
        M[j][k] += (T1[j], T2[k], A[i][j], ...)
      }
    }

    // Restore the storage layout:\\
    for (int j = 0; j<4; j+=4) {
      _m256d r0, r1, r2, r3, r4, r5, r6, r7;
      for (int k = 0; k<8; k+=4) {
        r0 = _mm256_load_pd (&A[j+0][k]);
        // Load A[j+1][k], A[j+2][k], A[j+3][k]
        r4 = _mm256_unpackhi_pd (r1, r0);
        r5 = _mm256_unpacklo_pd (r0, r1);
        r6 = _mm256_unpackhi_pd (r2, r3);
        r7 = _mm256_unpacklo_pd (r3, r2);
        r0 = _mm256_permute2f128_pd (r5, r7, 32);
        r1 = _mm256_permute2f128_pd (r4, r6, 32);
        r2 = _mm256_permute2f128_pd (r7, r5, 49);
        r3 = _mm256_permute2f128_pd (r6, r4, 49);
        mm256_store_pd (&A[j+0][k], r0);
        // Store M[j+1][k], M[j+2][k], M[j+3][k]
      }
    }
  }
}
```

Figure 3.10: Local assembly code for the Burgers problem in Figure 3.8(a) after the application of vector-register tiling (outer-product vectorization). In this example, the unroll-and-jam factor is 1.
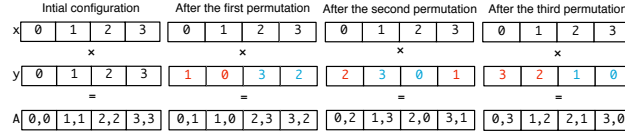
Figure 3.11: Outer-product vectorization by permuting values in a vector register.

matrix is abstractly expressible as

$$A_{jk} = \sum_{\substack{x \in B' \subseteq B \\ y \in B'' \subseteq B}} x_j \cdot y_k \qquad j, k = 0, ..., 2 \tag{3.8}$$

where B is the set of all basis functions (or temporary variables, e.g., `LI_0`) 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 3.11 illustrates how we can evaluate 16 entries ($j, k = 0, ..., 3$) of the element matrix using just 2 vector registers, which represent a 4×4 tile, assuming $|B'| = |B''| = 1$. Values in a register are shuffled each time a product is performed. Standard compiler auto-vectorization for both GNU and Intel compilers, instead, executes 4 broadcast operations (i.e., "splat" of a value over all of the register locations) along the outer dimension to perform the calculation. In addition to incurring a larger number of cache accesses, it needs to keep between $f = 1$ and $f = 3$ extra registers to perform the same 16 evaluations when unroll-and-jam is used, with f being the unroll-and-jam factor.

The storage layout of A, however, is incorrect after the application of this outer-product-based vectorization (*op-vect*, in the following). It can be efficiently restored with a sequence of vector shuffles following the pattern highlighted in Figure 3.12, executed once outside of the `ijk` loop nest. The pseudo-code for the Burgers local assembly kernel when using *op-vect* is shown in Listing **??**.

Initial configuration				After the first set of shuffles				After the second set of shuffles				Layout restored			
(0, 0)	(1, 1)	(2, 2)	(3, 3)	(0, 0)	(1, 1)	(2, 2)	(3, 3)	(0, 0)	(0, 1)	(2, 2)	(2, 3)	(0, 0)	(0, 1)	(0, 2)	(0, 3)
(0, 1)	(1, 0)	(2, 3)	(3, 2)	(0, 1)	(1, 0)	(2, 3)	(3, 2)	(1, 0)	(1, 1)	(3, 2)	(3, 3)	(1, 0)	(1, 1)	(1, 2)	(1, 3)
(0, 2)	(1, 3)	(2, 0)	(3, 1)	(0, 2)	(1, 3)	(2, 0)	(3, 1)	(0, 2)	(0, 3)	(2, 0)	(2, 1)	(2, 0)	(2, 1)	(2, 2)	(2, 3)
(0, 3)	(1, 2)	(2, 1)	(3, 0)	(0, 3)	(1, 2)	(2, 1)	(3, 0)	(1, 2)	(1, 3)	(3, 0)	(3, 1)	(3, 0)	(3, 1)	(3, 2)	(3, 3)
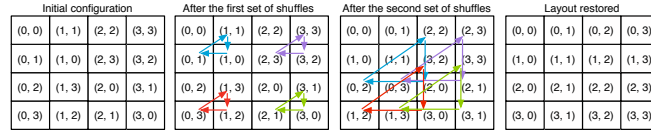
Figure 3.12: Restoring the storage layout after *op-vect*. The figure shows how 4×4 elements in the top-left block of the element matrix *A* can be moved to their correct positions. Each rotation, represented by a group of three same-colored arrows, is implemented by a single shuffle intrinsic.

3.5.4 Exposing Matrix-Matrix Multiplications for BLAS Operations

In this section, a way of systematically transforming the local element matrix computation into a sequence of matrix-matrix multiplication operations is discussed.

If such operations could be exposed, highly-optimized dense linear algebra libraries, for instance MKL or ATLAS BLAS, could be used, which would potentially result in notable performance improvements. It is true that the basis functions' size is usually too small to obtain any gain from BLAS routines, which are tuned for large arrays [Shin et al., 2010]; however, this can significantly increase with the order of the method and due to the presence of coefficient functions in the equation. Since our research is an exploration of optimization techniques for generic equations (i.e. nothing is assumed about the order of the method and the mathematical structure of the form), an algorithm capable of translating assembly expressions into a sequence of BLAS calls has been studied. The main steps of the algorithm are informally provided next.

By fully applying the rewrite rules in Figure 3.7, an assembly expression is reduced to a summation, over each quadrature point, of outer products along the test and trial functions. Each outer product is then isolated, i.e. the assembly expression is split into chunks, each chunk representing an outer product over test and trial functions. Statements in the bodies of the surrounding loops (e.g. coefficients evaluation at a quadrature point, temporaries introduced by expression rewriting) are vector-expanded and hoisted completely outside of the loop nest, similarly to what we have described in Section 3.4.3. This renders the loop nest perfect; that is, there is no

intervening code among the various loops. The element matrix evaluation has now become a sequence of dense matrix-matrix multiplies (transposition aside)

$$A_{jk} = \sum_i x_{0_{ij}} \cdot y_{0_{ik}} + \sum_i x_{1_{ij}} \cdot y_{1_{ik}} + \dots$$

where $x0, x1, y0, y1, \dots$ are tabulated basis functions or vector-expanded temporaries introduced at expression rewriting time. Eventually, the storage layout of the involved operands is changed so as to be conforming to the BLAS interface (e.g. two dimensional arrays are flatten as one dimensional arrays). The translation into a sequence of DGEMM calls is the last, straightforward step.

3.6 General-purpose Optimizations

3.6.1 Loop Interchange

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

3.6.2 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
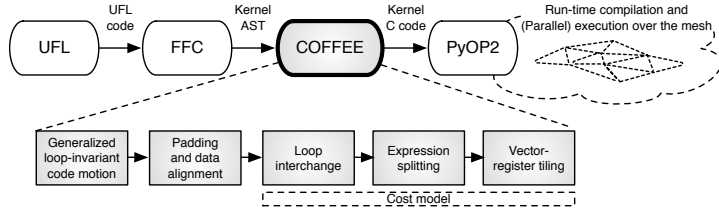
Figure 3.13: High-level view of Firedrake. COFFEE is at the core, receiving ASTs from a modified version of the FEniCS Form Compiler and producing optimized C code kernels.

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

By comparison with implementations characterized by manually-unrolled loops, we noticed that recent versions of compilers like GNU's and Intel's estimate close-to-optimal unroll factors when the loops are affine and their bounds are relatively small and known at compile-time, which is the case of our kernels. Unless an auto-tuning system is used (this option is discussed in Section 3.7.4) our choice is to leave the backend compiler in charge of applying loop unroll.

64

3.7 COFFEE: an Optimizing Compiler for Finite Element Local Assembly

3.7.1 Outline and Integration with Firedrake

Our research resulted in the implementation of COFFEE[2], a mature, platform-independent compiler capable of optimizing any local assembly code generated through Firedrake or FEniCS. COFFEE has been fully integrated with Firedrake, so it can optimize any equation expressible with this framework. The code, which comprises more than 5000 lines, is available at [**?**].

Firedrake users employ the Unified Form Language to express problems in a notation resembling mathematical equations. At run-time, the high-level specification is translated by a modified version of its form compiler, the FEniCS Form Compiler (FFC) Kirby and Logg [2006], into an abstract syntax tree (AST) representation of one or more finite element assembly kernels. ASTs are then passed to COFFEE to apply the transformations described in the previous sections. The output of COFFEE, C code, is eventually provided to PyOP2 [Markall et al., 2013], where just-in-time compilation and execution over the discretized domain take place. The flow and the compiler structure are outlined in Figure 3.13.

The compiler applies an ordered sequence of optimization steps to the ASTs received from FFC. Expression rewriting is performed first, due to the introduction of temporary arrays and the need to create multiple loop nest starting from the original one (this may happen when computation over zero-valued blocks is avoided). Then, code specialization takes place. Finally, loop interchange and loop unroll are applied. COFFEE has an auto-tuning system, described in Section 3.7.4, to individuate the most suitable optimization strategy (i.e. to select which transformations to apply) for a certain equation. The implementation of all code transformations is centered on analysis and manipulation of the kernel AST. Any possible corner cases are handled: for example, if outer-product vectorization is to be applied but the size of the iteration space is not a multiple of the vector length, then a remainder loop, amenable to auto-vectorization, is inserted (as shown in Figure **??**).

Users can either enable individual transformations by switching specific

---

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

flags or leave the auto-tuning system, described in Section 3.7.4, in charge of determining the best optimization strategy (i.e. selecting and composing transformations).

3.7.2 Modifying the FEniCS Form Compiler

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

Generation of Abstract Syntax Trees

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

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

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

bolic execution to avoid iteration over zero-valued blocks, as elaborated in Section 3.4.6.

To make FFC output ASTs, its intermediate representation is intercepted prior to the code generation stage and forwarded to a new module, which is referred to as `ast-generation`. The structure of the modified version of FFC is illustrated in Figure **??**. The `ast-generation` stage "mirrors" the actions of code generation, but it differs by rather building an AST. An example is provided in Figure **??**. To achieve this, COFFEE exposes its hierarchy of AST nodes, which are suitably instantiated and composed in `ast-generation`.

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

3.7.3 On the Compiler Implementation

Figure **??** outlines the various modules composing COFFEE. Expression rewriting, code specialization, and general-purpose transformations are applied by manipulating the AST representing the local assembly kernel. Providing the steps of all algorithms performing the various transformations would just be tedious and marginally helpful for the reader; essentially, implementing a transformation always reduces to write a routine that visits and manipulates a tree data structure (the AST) according to the semantics described in Sections 3.4 and 3.5. In this section, we rather focus on aspects of the compiler implementation that involve the orchestration of the different transformations, including a description of the data structures employed to track both the restructuring of the assembly expressions and data dependencies.

Code Hoisting

Hoisting is the operation of moving code, for instance a statement or the evaluation of an expression, from an inner to an outer loop in a nest. This operation is performed in several occasions when rewriting an expression:

generalized loop-invariant code motion, expansion of sub-expressions, and precomputation of terms all require code hoisting, as illustrated in Figures **??** and 3.6. In order to perform code hoisting, in particular, several aspects must be known:

1. "what" to hoist

2. "where" to hoist (possibly, outside of the loop nest)

3. if scalar-expansion should be introduced

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

- a reference to the AST node corresponding to the expression **e** hosted by **v**;

- a reference to the loop in which **v** is assigned **e**;

- a reference to the AST node corresponding to the declaration of **v**.

As expressions are hoisted, the dictionary is populated and/or updated with new information. For example, when applying generalized loop-invariant code motion, a new entry is created, unless the sub-expression being lifted has already been pre-computed elsewhere. On the other hand, when sub-expressions are expanded, either a new entry is created or an old entry is updated, as explained in Section 3.4.4.

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

Benefits of Tracking Data Dependencies

COFFEE implements "smart" code hoisting: everytime a sub-expression or a term are lifted, for example from the mathematical expression evaluating

the local element matrix to an outer level in the loop nest, three optimizations are potentially applied. Consider a hoistable expression `e` assuming different values in the iteration space `I`:

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

- **Loop fusion.** If scalar-expansion is introduced (we recall this is useful to achieve SIMD auto-vectorization; see Sections 3.4.1 and 3.4.3), then the hoisted code must be placed in an outer loop `r`, `r` $\in$ `I`. This was the purpose of the second `r` loop in Figure **??**; note that this loop includes scalar-expanded sub-expressions that, in the non-transformed code, iterated along logically different spaces (loops `j` and `k`, corresponding to test and trial functions). In this example, it was possible to use a single `r` loop because we assumed that test and trial function spaces were the same, leading to identical loop bounds; in general, however, this is not true. Another assumption of the example was that the space of the equation's coefficients (variables `f0, f1` in the figure) coincided with that of test and trial functions. This would allow fusing the two `r` loops, which is exactly what COFFEE does, although not displayed by the figure. Fusing loops, which increases data locality and reduces loop overhead, is possible in some circumstances, in particular when there are no data dependencies among hoisted expressions and the spaces of test, trial, and coeffcient functions are identical. As explained next, COFFEE reasons about data dependencies and iteration spaces to determine the safeness of loop fusion.

- **Reduce extra memory.** When expanding an expression, terms hoisting is possible to relieve register pressure. This poses the challenge described in Section 3.4.4 and intuitively summarized in Figure 3.6; that is, understanding whether it is possible to absorb the hoistable term in an available temporary value or a new temporary is needed. Obviously, the less is the number of temporaries introduced,

the smaller is the size of the working set, which may result in better performance.

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

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

3.7.4 Model-driven Dynamic Autotuning

Determining the sequence of transformations that maximizes the performance of a problem requires investigating a broad range of factors, including mathematical structure of the input form, polynomial order of employed function spaces, presence of pre-multiplying functions, and, of course, the characteristics of the underlying architecture. The set of possible optimizations is very large, so the selection problem is challenging. The sole Expression Rewriter, for instance, can generate a wide variety of implementations by just applying rewrite rules up to different extents.

We tackle the optimizations selection problem by compiler autotuning. Not only does it allow to determine the best combination of transformations, also it enables exploring parametric low-level optimizations, such as loop unroll, unroll-and-jam, and interchange, by trying different unroll factors and loop permutations. By leveraging domain knowledge and a set of heuristics, we manage to keep the autotuner overhead to a minimum, whilst achieving significant speed ups over a fixed optimization strategy, as shown in Section 3.8.3. In particular, our autotuner usually requires order of seconds to determine the fastest kernel implementation, a negligible overhead when it comes to iterate over real-life unstructured meshes; details will be

provided later.

COFFEE analyzes the input problem and decides what variants it is worth testing through autotuning, as described later. Each variant is obtained by requesting specific transformations to the Expression Rewriter and the Code Specializer. The possible variants are then provided to the autotuner, in the form of abstract syntax trees. The autotuner is a template-based code generator. By inspecting an abstract syntax tree, it determines how to generate "wrapping" code that 1) initializes kernel's input variables with fictitious values and 2) calls the kernel. These two points are executed repeatedly in a *while* loop for a pre-established amount of time (order of milliseconds). At the exit of the *while* loop, the times the kernel was invoked is recorded. Eventually, the variant executed the largest number of iterations is designated as the fastest implementation. Suitable compiler directives are used to prevent inlining of all function calls: this avoids the situation in which some variants are inlined and some are not, which would fake the autotuner's output.

The autotuning process is dynamic: depending on the complexity of the input problem, more or less variants are tried. General heuristics, which can be considered a revisited version of those presented in Shin et al. [2010], are applied:

- Loop permutations that are likely to worsen the performance are excluded from the search space. According to what explained in Section 3.6.1, we enable only variants in which the loop over quadrature points is either the outermost or the innermost. This is due to the fact that versions of the code in which such loop lies between the test and trial functions loops are typically lower performing.

- The unroll factors must divide the loop bounds evenly to avoid the introduction of reminder (scalar) loops.

- The innermost loop is never explicitly unrolled. This is because we expect auto-vectorization along this loop, so memory accesses should be kept unit-stride.

- Padding and data alignment are always applied.

The auto-tuning system is domain-aware; that is, it exploits the following heuristics, which capture properties of the computational domain:

- The length of test and trial functions loops are identical in some cases, for example when they originate from the same function space. In such cases, since for the employed storage layout the memory access pattern is symmetrical along these two loops, we prune their interchange from the search space.

- The larger is the polynomial order of the method, the larger is the assembly loop nest. In these cases, we impose a bound on the loop nest's overall unroll factor (which we found empirically) to avoid uselessly testing too many unroll factors.

- On the other hand, if the polynomial order is low, i.e. when the loop nest is small, we prune variants that we know will be low-performing, e.g. those resorting to BLAS.

- We specifically select two levels of expression rewriting. In the "base" level, only generalized loop-invariant code motion is applied. This means that only a subset of the rewrite rules exposed in Section 3.7 will be considered. In the "aggressive" level, all of the rewrite rules are applied, which means terms factorization, precomputation, and expansion (Sections 3.4.2-3.4.4) are introduced. Many other trade-offs, which we do not explore, would be feasibile, however.

- For the expression splitting optimization described in Section 3.5.2, only three split factors are tested, namely 1, 2, 4. Also, if the input problem uses vector-valued function spaces, the iteration space is already split at expression rewriting time to avoid computation over zero-valued columns; in such a case, we do not further apply expression splitting.

All of the previous points contribute to minimize the overhead of the autotuning process. We will discuss the actual cost of autotuning in Section 3.8.3.

3.8 Performance Analysis

3.8.1 Experimental Setup

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

Three studies are presented. Firstly, in Section 3.8.2, the impact of individual code transformations is analyzed in three real-world, representative equations. We believe it is important to clarify *in which contexts* and *why* such optimizations can provide speed ups. Secondly, we show that combining aggressive expression rewriting and code specialization allows outperforming both (i) a basic optimization strategy (where only a few of COFFEE's transformations are enabled) and (ii) the FEniCS Form Compiler's built-in optimization system. Experiments are carried out in a set of notable equations. In these first two studies, the speedups achievable at the level of the local assembly kernel are quantified; that is, the other stages of the computation (e.g. global assembly, problem setup) are excluded from the measurements. Finally, in Section 3.8.4, a full application investigation is provided to demonstrate that our techniques for cross-loop optimization of arithmetic intensity allow a significant reduction in the overall computation run-time to be achieved, which ultimately demonstrates the success of our research.

3.8.2 Contribution of Individual Optimizations

Rationale of the Study

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

- Generalized loop-invariant code motion (Section 3.4.1)

- Padding and data alignment (Section 3.5.1)

- Expression splitting (Section 3.5.2)

- Vector-register tiling (Section 3.5.3)

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

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

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

The benchmarks were written in UFL (code available at [Luporini, 2014c]) and executed over real unstructured meshes through Firedrake. The Helmholtz code has already been shown in Listing 6. 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 7), 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 $\mathbf{j}$ and $\mathbf{k}$ 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 $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$, and, as a consequence, the size of X and Y arrays). A range of polynomial orders from $q = 1$ to $q = 4$ were tested; higher polynomial orders are excluded from the study because of current Firedrake limitations. In all these cases, the size of the element matrix rarely exceeds 30×30, with a peak of 105×105 in Burgers with prisms and $q = 4$.

The speed ups due to applying optimizations over the original local assembly code are shown in Figure 3.14. The figure itself can be read as a "plot of plots": the various element shapes are reported on the horizontal axis, whereas each equation corresponds to one point along the vertical axis. This allows to deduct the behaviour of transformations when varying the parameters of our study in an unified way. In the next sections, we refer to this figure and elaborate on the impact of individual transformations. We shorten generalized loop-invariant code motion as *licm*; padding and data alignment as *ap*; outer-product vectorization as *op-vect*; expression splitting as *split*.

Impact of Generalized Loop-invariant Code Motion

In general, the speedups achieved by *licm* are notable. The main reasons were anticipated in Section 3.4.1: in the original code, 1) sub-expressions invariant to outer loops are not automatically hoisted, while 2) sub-expressions
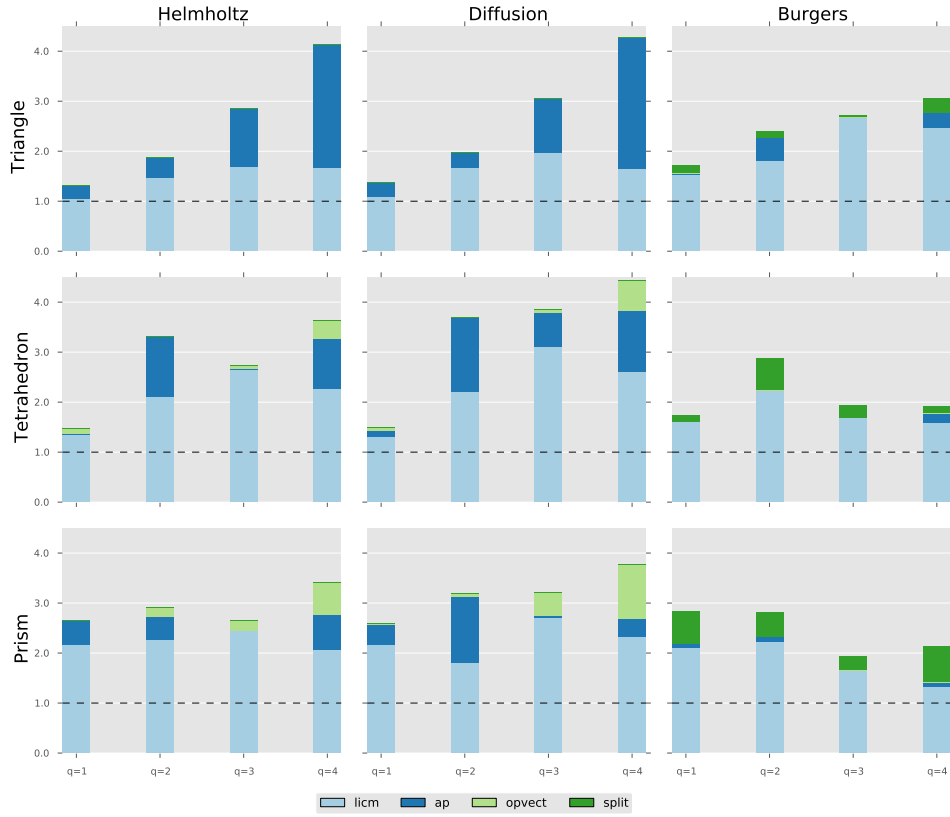
Figure 3.14: Performance improvement due to generalized loop-invariant code motion (*licm*), data alignment and padding (*ap*), outer-product vectorization (*op-vect*), and expression splitting (*split*) over the original non-optimized code. In each plot, the horizontal axis reports speed ups, whereas the polynomial order q of the method varies along the vertical axis.

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

Impact of Vector-register Tiling

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

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

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

The performance of individual kernels in terms of floating-point operations per second was also measured. The theoretical peak on a single core, with the Intel Turbo Boost technology activated, is 30.4 GFlop/s. In the

case of Diffusion using a prismatic mesh and $q = 4$, we achieved a maximum of 21.9 GFlop/s with *op-vect* enabled, whereas 16.4 GFlop/s was obtained when only *licm-ap* is used. This result is in line with the expectations: analysis of assembly code showed that, in the `jk` loop nest, which in this problem represents the bulk of the computation, 73% of instructions are actually floating-point operations.

Application of *op-vect* to the Burgers problem induces significant slow-downs 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 3.5.2, the price to pay for this optimazion is an increased number of accesses to the element matrix and, potentially, redundant computation.

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

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

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

are in line with the speedups shown in Figure 3.14.

The *split* transformation was also tried in combination with *op-vect* (*split-op-vect*). Despite improvements up to $1.22\times$, *split-op-vect* never outperforms *split*. This is motivated by two factors: for small split factors, such as 1 and 2, the data space to be tiled is still too big, and register spilling affects run-time; for higher ones, sub-expressions become so small that, as explained in Section 3.8.2, extensive unrolling already allows to achieve a certain degree of register re-use.

Results on Other Architectures

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

3.8.3 Evaluation in Forms of Increasing Complexity

Rationale of the Study

We analyze the run-time performance of four fundamental real-world problems, which comprise the differential operators that are most common in finite element methods. In particular, our study includes problems based upon the Helmholtz and Poisson equations, as well as elasticity- and hyperelasticity-like forms. The Unified Form Language Alnæs et al. [2014] specification for these forms, which is the domain specific language that both Firedrake and FEniCS use to express weak variational form, is available at [Luporini, 2014a].

We evaluate the *speed ups* achieved by three sets of optimizations over the original code; that is, the code generated by the FEniCS Form Compiler when no optimizations are applied. In particular, we analyze the impact of the FEniCS Form Compiler's built-in optimizations (henceforth `ffc`), the impact of COFFEE's transformations as presented in Section 3.8.2 (referred to as `fix`, in the following), and the impact due to COFFEE auto-tuning

expression rewriting and code specialization (henceforth `auto`, to denote the use of autotuning). The `auto` values do not include the auto-tuning overhead, which is commented aside in Section 3.8.3. All codes were executed in the context of the Firedrake framework.

The values that we report include the cost of local assembly as well as the cost of matrix insertion. However, the (unstructured) mesh was made small enough to fit the L3 cache of the architecture, so as to minimize the overhead due to any operations that are not part of the element matrix evaluation itself. We also reinforce the concept that the cost of local assembly becomes increasingly dominant in the whole finite element calculation as the complexity of the form increseas (e.g. [Olgaard and Wells, 2010]).

We do not compare to the FEniCS Form Compiler's tensor contraction mode [Kirby et al., 2005] because of three reasons: first, in [Olgaard and Wells, 2010] it has been demonstrated the superiority of quadrature as the complexity of a form increases, so it would be superfluous to repeat the same analysis. Second, our aim is to show the effect of low-level optimizations on the code, especially SIMD vectorization, which is not feasible in tensor mode due to the inherent structure of the code. Third, tensor mode code generation fails due to hardware limitations in many of the test cases that we show below.

We vary several aspects of each form, which follows the approach and the notation of [Olgaard and Wells, 2010] and [Russell and Kelly, 2013]

- The polynomial order of basis functions, $q \in \{1, 2, 3, 4\}$

- The polynomial order of coefficient (or "pre-multiplying") functions, $p \in \{1, 2, 3, 4\}$

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

On the other hand, other aspects are fixed

- The space of both basis and coefficient functions is Lagrange

- The mesh is three-dimensional, made of tetrahedrons, for a total of 4374 cells

Figures 3.15, 3.16, 3.17, and 3.18, which will be deeply commented in the next sections, must be read as "plots, or grids, of plots". Each grid

(figure) has two logical axes: p varies along the horizontal axis, while q varies along the vertical axis. The top-left plot in a grid shows speed ups for $[q = 1, p = 1]$; the plot on its right does the same for $[q = 1, p = 2]$, and so on. The diagonal of the grid shows plots for which basis and coefficient functions have same polynomial order, that is $q = p$. Therefore, a grid can be read in many different ways, which allows us to make structured considerations on the effect of the various optimizations.

A plot reports speed-ups over non-optimized FEniCS-Form-Compiler-generated code. There are three groups of bars, each group referring to a particular version of the code (`ffc, fix, auto`). There are four bars per group: the leftmost bar corresponds to the case $nf = 0$, the one on its right to the case $nf = 1$, and so on.

Results of General Applicability

The four chosen forms allow us to perform an in-depth evaluation of different classes of optimizations for local assembly. We limit ourselves to analyzing the cost of computing element matrices, although all of the techniques presented in this paper are immediately extendible to the evaluation of local vectors. As anticipated, in the following we comment speed ups of `ffc`, `fix`, and `auto` over the non-optimized, FEniCS-Form-Compiler-generated code.

We first comment on results of general applicability. By looking at the various figures, we note there is a trend in the optimizations to become more and more effective as q, p, and nf increase. This is because most of the transformations applied aim at optimizing for arithmetic intensity and SIMD vectorization, which obviously have a strong impact when arrays and iteration spaces are large. The corner cases of this phenomenon are indeed $[q = 1, p = 1]$ and $[q = 4, p = 4]$. We also observe how `auto`, in almost all scenarios, outperforms all of the other variants. In particular, it is not a surprise that `auto` is faster than `fix`, since `fix` is one of the autotuner's tested variants, as explained in Section 3.7.4. The reasons for which `auto` exceeds both original code and `ffc` are discussed for each specific problem next. Also, details on the "optimal" code variant determined by autotuning are given in Section 3.8.3.
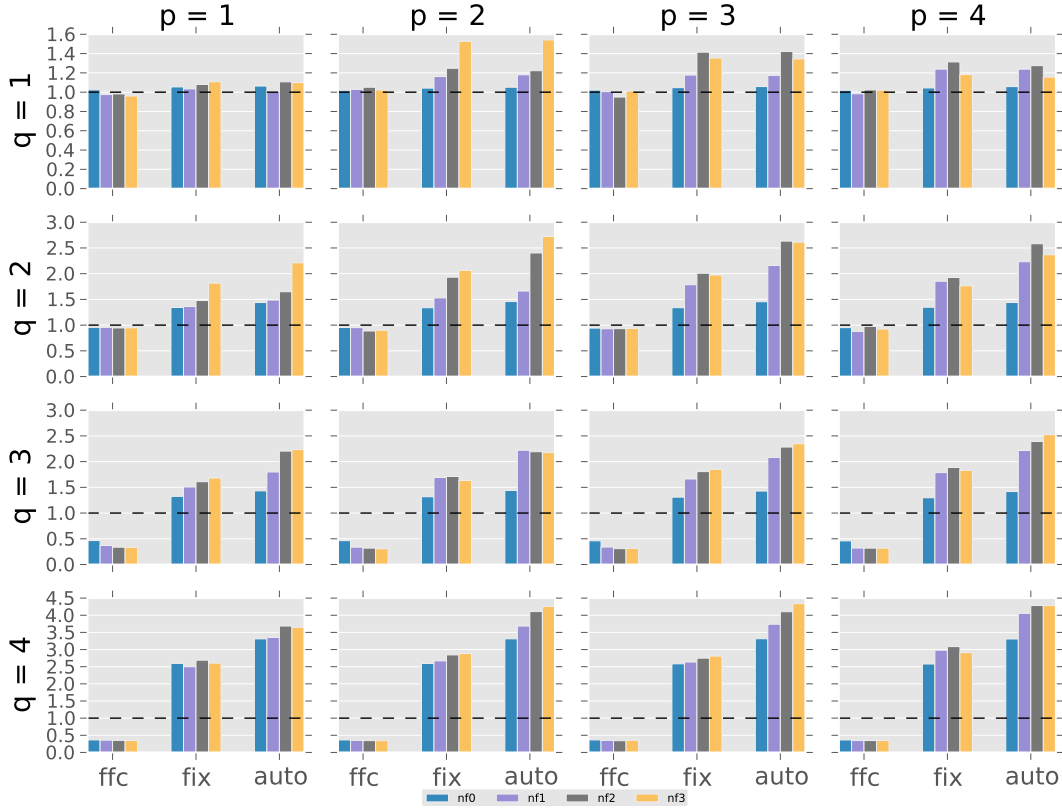
Figure 3.15: Helmholtz results.

Performance of Helmholtz

The results for the Helmholtz problem are provided in Figure 3.15. We observe that `ffc` slows the code down, especially for $q \geq 3$. This is a consequence of using indirection arrays in the generated code that, as explained in Section 3.4.6, prevent, among the other compiler's optimizations, SIMD auto-vectorization. The `auto` version results in minimal performance improvements over `fix` when $nf = 0$, unless $q = 4$. This is due to the fact that if the loop over quadrature points is relatively small, then close-to-peak performance is obtainable through basic expression rewriting and code specialization; in this circumstance, generalized loop-invariant code motion and padding plus data alignment. The trend changes dramatically as nf and q increase: a more ample spectrum of transformations must be considered to find the optimal local assembly implementation. We will provide details about the selected transformations in the next section.
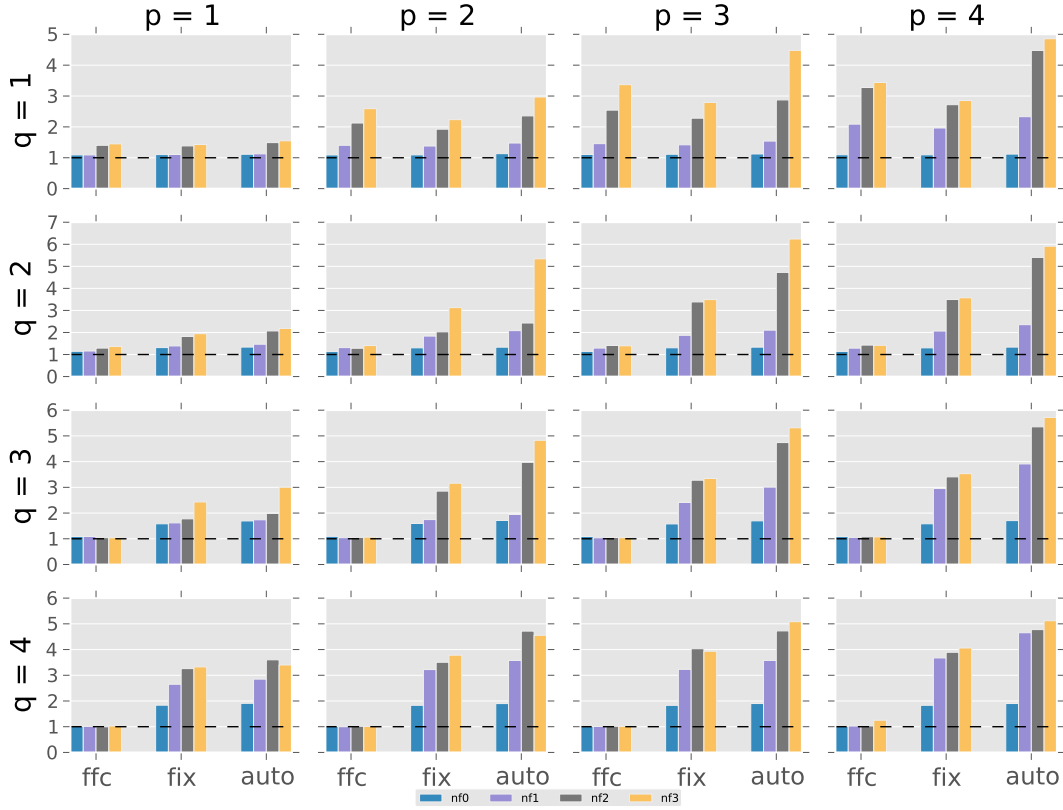
Figure 3.16: Elasticity results.

Performance of Elasticity

Figure 3.16 illustrates results for the Elasticity problem. This form uses a vector-valued space for the basis functions, so here transformations avoiding computation over zero-valued columns are of key importance. The `ffc` set of optimizations leads to notable improvements over the original code at $q = 1$. The use of inderection arrays allows to phisically eliminate zero-valued columns at code generation time; as a consequence, different tabulated basis functions are merged into a single array. Therefore, despite the execution being purely scalar because of indirection arrays, the reduction in arithmetic intensity and register pressure imply improvement in performance. Nevertheless, `auto` remains in general the best choice, with gains over `ffc` that are wider as p and nf increase.

For $q \geq 2$, in `ffc` the lack of SIMD vectorization counterbalances the decrease in the number of floating point operations, leading to speed ups
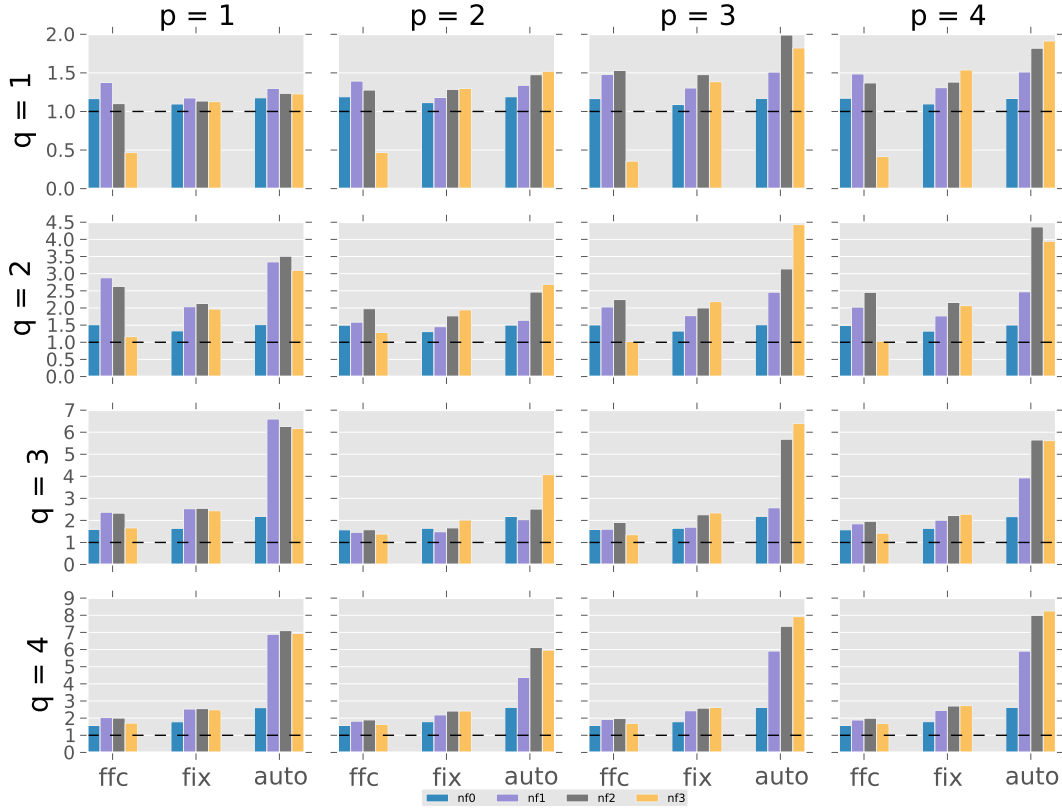
Figure 3.17: Poisson results.

over the original code that only occasionally exceed $1\times$. On the other hand, the successful application of the zero-avoidance optimization while preserving code specialization plays a key role for `auto`, resulting in much higher performance code especially at $q = 2$ and $q = 3$.

It is worth noting that speed ups of `auto` over `fix` decrease at $q = 4$, particularly for low values of p. As we will discuss in Section 3.8.3, this is because at $q = 4$ the vector-register tiling transformation (in combination with loop unroll-and-jam) leads to the highest performance. In principle, vector-register tiling can be used in combination to the zero-avoidance technique; however, due to mere technical limitations, this is currently not supported in COFFEE. Once solved, we expect much higher speed ups in the $q = 4$ regime as well.

Performance of Poisson

In Figure 3.17 we report speed ups of `ffc`, `fix`, and `auto` over the original code for the Poisson form. We note that, as a general trend, `ffc` exhibits drops in performance as nf increases, notably when $nf = 3$, for any values of q and p. This is a consequence of the inherent complexity of the generated code. The way `ffc` performs loop-invariant code motion leads to the pre-computation of integration-dependent terms at the level of the integration loop, which are characterized by higher arithmetic intensity and redundant computation as nf increases. Moreover, the absence of vectorization is another limiting factor.

The `auto` variant generally shows the best performance. Significant improvements over `fix` are also achieved, notably as q, p and nf increase. As clarified in the next section, this is always due to a more aggressive expression rewriting in combination with the zero-avoidance technique.

Performance of Hyperelasticity

Speed ups for the hyperelasticity form are shown in Figure 3.18. Experiments for $nf \geq 2$ could not be executed because of FEniCS-Form-Compiler's technical limitations.

For `auto`, massive speed ups for $q \geq 2$ are to be ascribed to aggressive and successful expression writing. Hyperelasticity problems are really compute-intensive, with thousands of operations being performed, so reductions in redundant and useless computation are crucial. Complex forms like hyperelasticity would benefit from further "specialized" optimizations: for example, it is a known technical limitation of COFFEE that, in some circumstances, less temporaries could (should) be generated and that hoisted code could (should) be suitably distributed over different loops to minimize register pressure (e.g. COFFEE could apply loop fission for obtaining significantly better register usage). We expect to obtain considerably faster code once such optimizations will be incorporated.

In the regime $q \geq 2$ and $nf = 1$, peformance improvements are less pronounced moving from $p = 1$ to $p = 2$, although still significant; in particular, we notice a drop at $p = 2$, followed by a raise up to $p = 4$. It is worth observing that this effect is common to all sets of optimizations. The hypothesis is that this is due to the way coefficient functions are evaluated
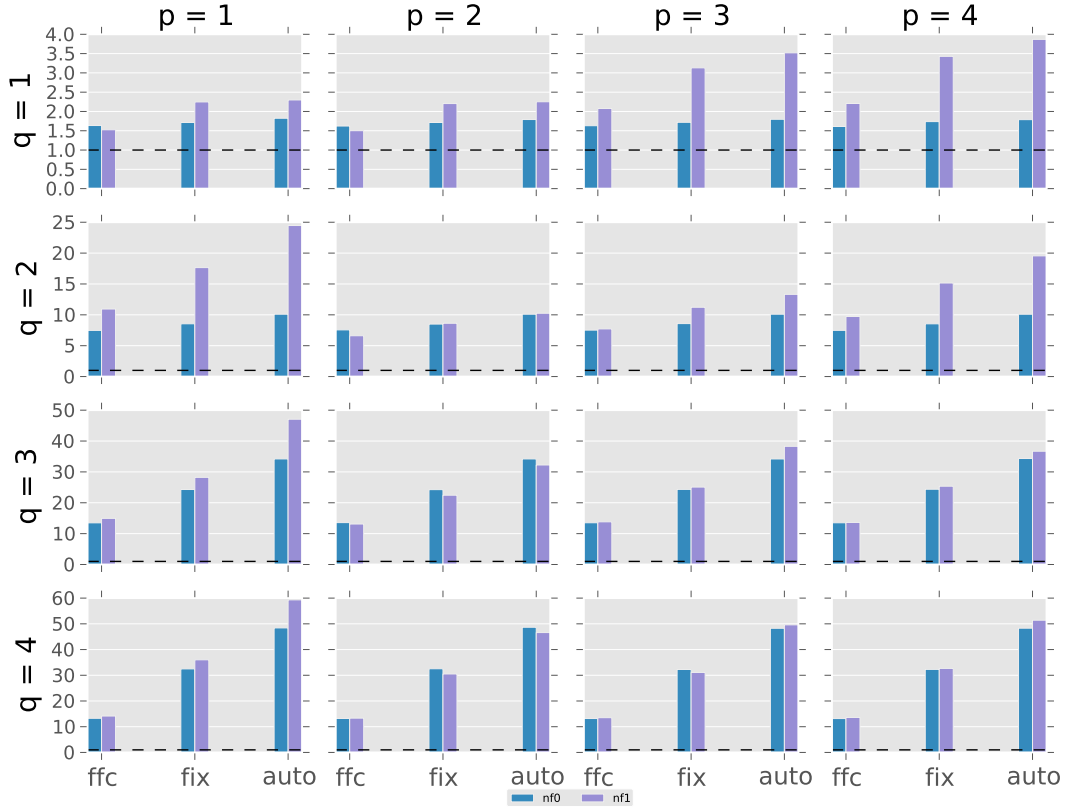
Figure 3.18: Hyperelasticity results.

at quadrature points (identical in all configurations), which cannot be easily vectorized unless a change in storage layout and loops order is implemented in the code (abstract syntax tree) generator on top of COFFEE.

Details on the Autotuning Process

Autotuning Overhead   We first comment on the overhead of the auto-tuning process. In the context of Firedrake, the framework in which COF-FEE is integrated, the autotuner is executed at run-time, once the local assembly kernels are provided by the FEniCS Form Compiler. Autotuning, therefore, introduces an overhead in the application execution time. Such overhead originates from four operations: 1) creation of the various code variants; 2) generation of a C file containing such variants (as simple function calls, plus the "main" function that invokes the variants, in sequence); 3) compilation of the autotuning file; 4) execution. Of these four points, we

note that: the cost of 4) is relatively small, because each variant's execution time is bound by an empirically-found value (e.g. some milliseconds). The cost of all four points is constrained by our heuristics to prune the search space, as described in Section 3.7.4. Moreover, for a given form and a given discretization, the autotuner needs to be executed only once, since its output is saved and reused for later assemblies. This implies that if the assembly occurs in a time stepping loop, or the same form is executed on a different mesh, or known quantities of the input problem are changed, then the assembly cost is rapidly amortized. Premised that, the most important thing remains that when working with real unstructured meshes - which are likely to be composed of millions of elements, leading to long-lasting assembly phases - the autotuner overhead practically becomes completely negligible. In our experience, and in particular in the four examined problems, the autotuning process lasted less than a minute in the majority of cases. Most often, it took less than thirty seconds. Rarely it needed more than a minute, as for instance in the case of hyperelasticity; however, despite the inherent complexity of this form, we measured a peak of only 4 minutes for the extreme case $[q = 4, p = 4, nf = 1]$, while 1.30 minutes were needed in the case $[q = 1, p = 1, nf = 1]$. This analysis certifies that the autotuner overhead is definitely sustainable in real-world applications.

Selected Optimizations   We have repeatedly claimed that different forms (and different discretizations) require distinct sets of transformations to reach close-to-peak performance. To demonstrate this, we now report details about the output of the autotuning process. We show that for the four examined forms - more specifically, for the 224 problem instances $[form, q, p, nf]$ illustrated in the previous figures - a plethora of optimization strategies have been selected by the autotuner. We also complement and strengthen our claim by showing the autotuner's output for two additional forms whose performance results, for brevity, were not shown in Section 3.8.3: a Mass and a Mixed Poisson problems.

Table 3.2 shows the number a given transformation has been selected by the autotuner. To not hinder readability, the output has been grouped by form, rather than showing the selected optimization strategy for each $[form, q, p, nf]$. Values in the `rewrite strategy` column can either be `base` or `aggressive`, as explained in Section 3.7.4: in the former case, only

		expression rewriting			code specialization				
problem	number of variants	rewrite strategy	zero-valued columns avoidance	precompute integration terms	padding alignment	split	vector register tiling	unroll	BLAS
helmholtz	64	aggressive	0	0	all	12	26	20	0
mass	64	base	0	0	all	0	0	31	2
elasticity	64	aggressive	39	0	all	0	11	0	2
poisson	64	aggressive	52	0	all	all	0	0	0
mixed poisson	192	base	0	0	all	32	68	36	0
hyperelasticity	32	aggressive	all	all	all	2	0	0	0

Table 3.2: Summary of the optimizations selected by the autotuner in a number of forms

generalized loop-invariant code motion in applied; in the latter case, the rewrite rules are recursively applied, as extensive as possible. If the value is `aggressive` (`base`) it means that in the majority of cases the `aggressive` (`base`) strategy prevailed on the `base` (`aggressive`) one. Precomputation of integration-dependent terms was explained in Section 3.4.3. The `split` column refers to the expression splitting transformation (Section 3.5.2). The `unroll` column indicates the application of explicit unrolling. Other columns are of obvious meaning. Values in the various cells illustrate the number of problem instances (out of the total reported in column `number of variants`) in which an optimization was activated. Obviously, more transformations are typically used in combination in a same problem.

3.8.4 Full Application Study

In this section, we investigate the performance gain for an entire finite element computation developed in Firedrake, a linear elasticity problem. The source code is available at [Luporini, 2014b]. The equation is used to simulate deformation of an object caused by pre-established loading conditions.

The execution time of a steady finite element problem is dominated by two factors: assembly and solve. The evaluation of all local element matrices and vectors, and their insertion, respectively, in a "global" sparse matrix and a global vector, compose the "assembly" phase. The global matrix and vector form a linear system, usually solved by an iterative method; this is the "solve" phase. The percentage of time spent in assembly and solve varies from problem to problem. As reiterated in the available literature, for example in ?, the computational cost of local assembly grows with the *complexity* of the partial differential equation (e.g. because of larger loops and more articulated expressions), while the solve time increases with polynomial order and mesh discretization. The complexity of an equation depends on several
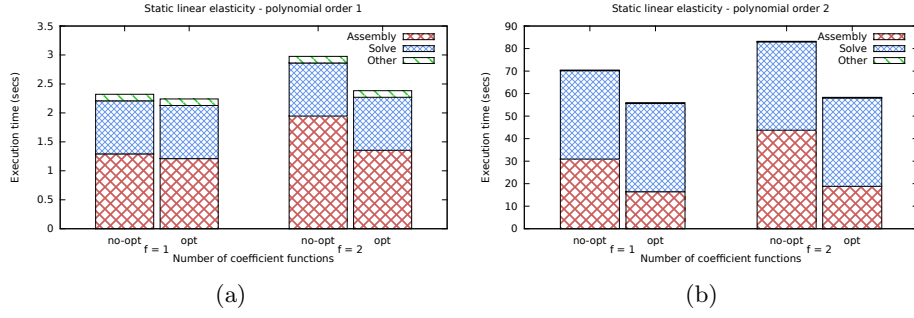
Figure 3.19: Performance improvement over non-optimized code for the static linear elasticity equation on a single core of the Sandy Bridge architecture.

factors, including the number of derivatives and coefficient functions (i.e. additional known functions that characterize the equation).

We study two versions of the static linear elasticity problem: 1) with one coefficient function, $f = 1$, and 2) with two coefficient functions, $f = 2$. More coefficient functions are also plausible; however, as we will see, the assembly process starts being particularly expensive already with $f = 1$. For each of these two versions, we examine the cases of polynomial order $q = 1$ and $q = 2$. To run our experiments, we use a single core of the Sandy Bridge architecture described in Section **??**. The application runs on a tetrahedral mesh composed of 196608 elements.

In Figure 3.19, we show the execution times for the four test cases, split over two figures (one for each polynomial order), without and with optimizations enabled. A stacked bar captures the time spent in the assembly phase (`assembly`), in solving the linear system (`solve`), and in the other various parts of the program (`other` - for example, setup of the problem and initialization of the coefficient functions). The *non-opt* and *opt* bars correspond, respectively, to the cases in which no optimizations and a combination of optimizations have been applied. The optimizations applied are generalized loop-invariant code motion, alignment and padding, and expression splitting. We recall that the cost of the insertion of the computed local element matrices (and vectors) in the global matrix (vector) is incorporated in `assembly`.

We first notice that, in the scenario ($f = 1, p = 1$), the assembly is dominated by matrix insertion: despite the application of several transfor-

mations, only a minimal performance gain is achieved. However, if either q or f is increased then the cost of `assembly` becomes larger with respect to `solve`, and the matrix insertion cost becomes negligible. In such cases, the transformations automatically applied by COFFEE dramatically decrease the cost of `assembly`. This results in a significant overall speedup and a simulation which is now dominated by `solve` time. It is interesting to note that generalized loop-invariant code motion was particularly invasive in this case, with 23 temporaries generated and several redundancies discovered.

In these experiments, we observe a maximum performance improvement of $1.47\times$ over the non-optimized local assembly code, obtained in the case $(f = 2, p = 2)$. However, we reiterate the fact that full application speed ups increase proportionally with the amount of time spent in assembly and, therefore, with the complexity of the equation. By increasing polynomial order and number of coefficient functions, or by simply studying a different, more complex equation, it is our experience that performance gains become increasingly more relevant. The choice of studying the static linear elasticity equation was to show that even relatively simple problems can be characterized by a large proportion of execution time spent in assembly.

3.9 Related Work

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

Optimization of quadrature-based local assembly for CPU architectures has been addressed in FEniCS Olgaard and Wells [2010]. The compari-

son between COFFEE and this work has been presented in Section 3.8.3. In [Markall et al., 2010], and more recently in [Knepley and Terrel, 2013], the same problem has been studied for GPU architectures. In [Kruel and Bana, 2013], variants of the standard numerical integration algorithm have been specialized and evaluated for the PowerXCell processor, but an exhaustive study from the compiler viewpoint - like ours - is missing, and none of the optimizations presented in this chapter are mentioned. Among these efforts, to the best of our knowledge, COFFEE is the first work targeting low-level optimizations through a real compiler approach.

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

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

Many code generators, like those based on the Polyhedral model Bondhugula et al. [2008] and those driven by domain-knowledge Stock et al. [2011], make use of cost models. The alternative of using auto-tuning to select the best implementation for a given problem on a certain platform has been adopted by nek5000 Shin et al. [2010] for small matrix-matrix multiplies, the ATLAS library Whaley and Dongarra [1998], and FFTW Frigo

and Johnson [2005] for fast fourier transforms. In both cases, pruning the implementation space is fundamental to mitigate complexity and overhead. Likewise, COFFEE uses heuristics and a model-driven auto-tuning system (Section 3.7.4) to steer the optimization process.

3.10 Generality of the Approach and Applicability to Other Domains

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

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

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

Generalizability to other domains There are neither conceptual nor technical reasons which prevent our compiler from being used in applications other than Firedrake. For example, integration with the popular FEniCS

framework, the pioneer of automated code generation for finite element local assembly, would be relatively easy to achieve. It is more challenging to assess the generality of the optimization strategy: the extent to which COFFEE and its transformations are transferable to other computational domains, perhaps other DSLs, and to what extent this would be helpful for improving full application performance. To answer these questions, we first need to go back to the origins of our compiler. The starting point of our work was the mathematical formulation of a local assembly operation, expressible as follows

$$\forall_{i,j} \quad A_{ij}^K = \sum_{q=1}^{n_1} \sum_{k=1}^{n_2} \alpha_{k,q}(a', b', c', ...) \beta_{q,i,j}(a, b, c, d, ...) \gamma_q(w_K, z_K) \tag{3.9}$$

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

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

  1. In Equation (3.9), 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 such as COFFEE could map mathematical quantifiers and operators to different parallelization schemes and generate distinct variants of multithreaded kernel code. Based on our experience, we believe this is the right approach to achieve performance portability.

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

3. Associative arithmetic operators are the prerequisite for expression splitting. In essence, this transformation concerns resource-aware execution. In the context of COFFEE, expression splitting has been successfully applied to improve register pressure. However, the underlying idea of re-scheduling (re-associating) operations to optimize for some generic parameters is far more general. It could be used, for example, as a starting point to perform kernel fission; that is, splitting a kernel into multiple parts, each part characterized by less stringent memory requirements (a variant of this idea for non-affine loops in unstructured mesh applications has been adopted in [?]). In Equation (3.9), 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

94

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 3.9, 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.

3.11 Conclusion

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

Bibliography

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

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

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

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

AMCG. *Fluidity Manual*. Applied Modelling and Computation Group, Department of Earth Science and Engineering, South Kensington Campus, Imperial College London, London, SW7 2AZ, UK, version 4.0-release edition, November 2010. available at `http://hdl.handle.net/10044/1/7086`.

Krzysztof Bana, Przemyslaw Plaszewski, and Pawel Maciol. Numerical integration on gpus for higher order finite elements. *Comput. Math. Appl.*, 67 (6):1319–1344, April 2014. ISSN 0898-1221. doi: 10.1016/j.camwa.2014. 01.021. URL `http://dx.doi.org/10.1016/j.camwa.2014.01.021`.

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

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`.

Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 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 contributors. The Firedrake Project. `http://www.firedrakeproject.org`, 2014.

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

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

Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2130-3. doi: 10.1145/2464996.2467268. URL `http://doi.acm.org/10.1145/2464996.2467268`.

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

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

Robert C. Kirby, Matthew Knepley, Anders Logg, and L. Ridgway Scott. Optimizing the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 27(3):741–758, October 2005. ISSN 1064-8275. doi: 10.1137/040607824. URL `http://dx.doi.org/10.1137/040607824`.

A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous galerkin methods on graphics processors. *J. Comput. Phys.*, 228(21):7863–7882, November 2009. ISSN 0021-9991. doi: 10.1016/j.jcp.2009.06.041. URL `http://dx.doi.org/10.1016/j.jcp.2009.06.041`.

Matthew G. Knepley and Andy R. Terrel. Finite element integration on gpus. *ACM Trans. Math. Softw.*, 39(2):10:1–10:13, February 2013. ISSN 0098-3500. doi: 10.1145/2427023.2427027. URL `http://doi.acm.org/10.1145/2427023.2427027`.

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

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

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

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

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

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

Fabio Luporini. Code for experiments on individual transformations in

COFFEE. `https://github.com/firedrakeproject/firedrake/tree/pyop2-ir-perf-eval/tests/perf-eval`, 2014c.

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

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

Graham R. Markall, David A. Ham, and Paul H.J. Kelly. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science*, 1(1):1815 – 1823, 2010. ISSN 1877-0509. doi: http://dx.doi.org/10.1016/j.procs.2010.04.203. URL `http://www.sciencedirect.com/science/article/pii/S1877050910002048`. ICCS 2010.

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

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

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

Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo.

SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93 (2):232– 275, 2005.

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

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

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

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

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

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

K. Stock, T. Henretty, I. Murugandi, P. Sadayappan, and R. Harrison. Model-driven simd code generation for a multi-resolution tensor kernel.

In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1058–1067, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4385-7. doi: 10.1109/IPDPS.2011.101. URL `http://dx.doi.org/10.1109/IPDPS.2011.101`.

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

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

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

M.M. Strout, F. Luporini, C.D. Krieger, C. Bertolli, G.-T. Bercea, C. Olschanowsky, J. Ramanujam, and P.H.J. Kelly. Generalizing 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`.

Peter E. J. Vos, Spencer J. Sherwin, and Robert M. Kirby. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *J. Comput. Phys.*, 229(13):5161–5181, July 2010. ISSN 0021-9991. doi: 10.1016/j.jcp.2010.03.031. URL `http://dx.doi.org/10.1016/j.jcp.2010.03.031`.

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