Published: 09 January 2015
Accepted: 01 October 2014
Revised: 01 October 2014
Received: 01 July 2014

RESEARCH-ARTICLE

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

FABIO LUPORINI, Imperial College London, London, U.K.

ANA LUCIA VARBANESCU, University of Amsterdam, Amsterdam, Noord-Holland, Netherlands

FLORIAN RATHGEBER, Imperial College London, London, U.K.

GHEORGHE-TEODOR BERCEA, Imperial College London, London, U.K.

J (RAM) RAMANUJAM, Louisiana State University, Baton Rouge, LA, United States

DAVID A HAM, Imperial College London, London, U.K.

View all

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

FABIO LUPORINI, Imperial College London
ANA LUCIA VARBANESCU, University of Amsterdam
FLORIAN RATHGEBER and GHEORGHE-TEODOR BERCEA, Imperial College London
J. RAMANUJAM, Louisiana State University
DAVID A. HAM and PAUL H. J. KELLY, Imperial College London

We study and systematically evaluate a class of composable code transformations that improve arithmetic intensity in local assembly operations, which represent a significant fraction of the execution time in finite element methods. Their performance optimization is indeed a challenging issue. Even though affine loop nests are generally present, the short trip counts and the complexity of mathematical expressions, which vary among different problems, make it hard to determine an optimal sequence of successful transformations. Our investigation has resulted in the implementation of a compiler (called COFFEE) for local assembly kernels, fully integrated with a framework for developing finite element methods. The compiler manipulates abstract syntax trees generated from a domain-specific language by introducing domain-aware optimizations for instruction-level parallelism and register locality. Eventually, it produces C code including vector SIMD intrinsics. Experiments using a range of real-world finite element problems of increasing complexity show that significant performance improvement is achieved. The generality of the approach and the applicability of the proposed code transformations to other domains is also discussed.

Categories and Subject Descriptors: G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Finite element methods*; G.4 [**Mathematical Software**]: Parallel and Vector Implementations

General Terms: Design, Performance

Additional Key Words and Phrases: Finite element integration, local assembly, compilers, optimizations, SIMD vectorization

ACM Reference Format:
Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. 2015. Cross-loop optimization of arithmetic intensity for finite element local assembly. ACM Trans. Architec. Code Optim. 11, 4, Article 57 (January 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2687415

This research is partly funded by the MAPDES project; the Department of Computing at Imperial College London; EPSRC through grants EP/I00677X/1, EP/I006761/1, and EP/L000407/1; NERC grants NE/K008951/1 and NE/K006789/1; the U.S. National Science Foundation through grants 0811457, 0926687, and 1059417; the U.S. Army through contract W911NF-10-1-000; Louisiana State University; and an HiPEAC collaboration grant.
Authors' addresses: F. Luporini, F. Rathgeber, G.-T. Bercea, and P. H. J. Kelly, Department of Computing, Imperial College London; email: {f.luporini12, f.rathgeber10, gheorghe-teodor.bercea08, p.kelly}@imperial.ac.uk; A. L. Varbanescu, Informatics Institute, University of Amsterdam; email: a.l.varbanescu@uva.nl; D. A. Ham, Department of Computing and Department of Mathematics, Imperial College London; email: david.ham@imperial.ac.uk; J. Ramanujam, Center for Computation and Technology and the School of Electrical Engineering and Computer Science, Louisiana State University; email: ram@cct.lsu.edu.

1. INTRODUCTION

In many fields, such as computational fluid dynamics, computational electromagnetics, and structural mechanics, phenomena are modeled by partial differential equations (PDEs). Numerical techniques, such as 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, as they allow an accurate representation of complex geometries. The solution is sought by applying suitable numerical operations, described by computational kernels, to all entities of a mesh, such as edges, vertices, or cells. On standard clusters of multicores, a kernel typically is executed sequentially by a thread, whereas 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, by Markall et al. [2013], Logg et al. [2012], AMCG [2010], and DeVito et al. [2011].

The time required to execute the numerical kernels is a major issue, because 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 article.

In particular, we address the well-known problem of optimizing the local assembly phase of the finite element method [Russell and Kelly 2013; Ølgaard and Wells 2010; Knepley and Terrel 2013; Kirby et al. 2005], which can be responsible for a significant fraction of the overall computation runtime, often in the range of 30% to 60%. With respect to these studies, we propose a novel set of composable code transformations targeting, for the first time, cross-loop arithmetic intensity, with emphasis on instruction-level parallelism, redundant computation, and register locality. In addition, we discuss how such transformations generalize to domains other than finite element local assembly.

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. In this work, we focus on relatively low order finite element methods, in which an assembly kernel's working set is usually small enough to fit in the L1 cache of traditional CPU architectures. 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.

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 for the execution on GPUs is the stringent memory requirements. Only relatively small problems fit in 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, which we mention in Section 7; these works differ from ours in several ways, including (1) not optimizing for cross-loop arithmetic intensity (the goal is rather effective multithread parallelization), and (2) not relying on automated code generation from a domain-specific language (DSL, explained next); moreover, they have done testing only on order 1 methods, whereas we focus on the order range of 1 to 4. In addition, our code transformations would drastically impact the GPU parallelization strategy, such as by increasing a thread's working set. For all of these reasons, a study on extending our work to GPU architectures is beyond the scope of this article. In Section 6, however, we provide some intuitions about this research direction.

An assembly kernel is characterized by the presence of an affine, often nonperfect 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 2D array, representing the result of local assembly in an element of the discretized domain. With such kernel and working set structures, traditional transformations for cache locality (e.g., blocking) are not helpful; instead, we focus on aspects such as the optimization of floating-point operations, register locality, and instruction-level parallelism, especially vectorization.

Achieving high performance on CPUs is nontrivial. The complexity of the mathematical expressions, often characterized by a large number of operations on constants and small matrices, makes it hard to determine an optimal sequence of transformations that is successfully applicable to different problems (equations). Loop trip counts are typically small and can vary significantly, which further exacerbates the issue. Therefore, a compiler-based approach is the only reasonable option to obtain close-to-peak performance in a wide range of different local assembly kernels. Optimizations like padding, generalized loop-invariant code motion, vector-register tiling, and expression splitting, as well as their composition, are essential, but their support in state-of-the-art vendor and polyhedral compilers, if any, is quite limited. BLAS routines could theoretically be employed, although a fairly complicated control and dataflow analysis would be required to automate identification and extraction of matrix-matrix multiplications. In addition, as detailed in Section 5.2.8, the small dimension of the matrices involved and the potential loss in data locality can limit or eliminate the performance gain of this approach.

To overcome the constraints of the available compilers and specialized linear algebra libraries, we have automated a set of generic and model-driven code transformations in COFFEE,[1] a compiler for optimizing local assembly kernels. COFFEE is integrated with [Firedrake 2014], a system for solving PDEs through the finite element method based on the PyOP2 abstraction [Rathgeber et al. 2012; Markall et al. 2013]. All problems expressible with this framework are supported by COFFEE, including equations that can be found at the core of real-world simulations, such as those used in our performance evaluation. In evaluating our code transformations for a range of relevant problems, we vary two key parameters that impact solution accuracy and kernel cost: the polynomial order of the method (we investigate from $p = 1$ to $p = 4$) and the geometry of elements in the discretized domain (2D triangle, 3D tetrahedron, 3D prism). From the point of view of the generated code, these two parameters directly impact the size of both loop nests and mathematical expressions, as elaborated in Section 5.2.1.

Our experiments show that the original generated code for nontrivial assembly kernels, despite following state-of-the-art techniques, remains suboptimal in the context of modern multicore architectures. Our domain-aware cost model–driven sequence of code transformations, aimed at improving SIMD vectorization and register data locality, can result in performance improvements up to 4.4× over original kernels. Around 70% of the test cases obtain a speedup greater than 2×.

[1]COFFEE stands for COmpiler For Fast Expression Evaluation.

```
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
```

Fig. 1. Structure of a local assembly kernel.

Summarizing, the contributions of this work are as follows:

(1) An optimization strategy for finite element local assembly that exploits domain knowledge and goes beyond the limits of both vendor and research compilers
(2) The design and implementation of a compiler that automates the proposed code transformations for any problems expressible in Firedrake
(3) A systematic analysis using a suite of examples of real-world importance that is evidence of significant performance improvements on two Intel architectures, a Sandy Bridge CPU and the Xeon Phi
(4) A full application evaluation to demonstrate that the optimization strategy greatly impacts the whole finite element problem.

Our contributions also include a theoretical study involving

(5) An analysis of the generality of the proposed code transformations and an investigation of their applicability to different computational domains.

The article is organized as follows. In Section 2, we provide some background on local assembly, show code generated by Firedrake, and emphasize the critical computational aspects. Section 3 describes the various code transformations, highlighting when and how domain knowledge has been exploited. The design and implementation of our compiler is discussed in Section 4. Section 5 shows performance results. The generality of our optimization strategy and the applicability to other domains is discussed in Section 6. Related work is described in Section 7. Finally, Section 8 reviews our contributions in the light of our results and identifies priorities for future work.

2. BACKGROUND AND MOTIVATING EXAMPLES

Local assembly is the computation of contributions of a specific cell in the discretized domain to the linear system that yields the PDE solution. The process consists of numerically evaluating problem-specific integrals to produce a matrix and a vector [Ølgaard and Wells 2010; AMCG 2010], whose sizes depend on the order of the method. This operation is applied to all cells in the discretized domain. In this work, we focus on local matrices or *element matrices*, which are more costly to compute than element vectors.

Given a finite element description of the input problem expressed through the domain-specific Unified Form Language (UFL) [Alnæs et al. 2014], Firedrake employs the FEniCS form compiler (FFC) [Kirby and Logg 2006] to generate an abstract syntax tree (AST) of a kernel implementing assembly using numerical quadrature. This kernel can be applied to any element in the mesh, which follows from a mathematical property of the finite element method. The evaluation of a local matrix can be reduced to integration on a fixed *reference* element—a special element that does not belong to the domain—after a suitable change of coordinates. Firedrake triggers the compilation of an assembly kernel using an available vendor compiler and manages its parallel execution over all elements in the mesh. As already explained, the subject of this article is to enhance this execution model by adding an optimization stage prior to the generation of C code.

The structure of a local assembly kernel is shown in Figure 1. The inputs are a zero-initialized 2D array used to store the element matrix, the element's coordinates in the

Table I. Type and Variable Names Used in the Various Listings to Identify Local Assembly Objects

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
Numerical integration weights	double[]	W
Basis functions (and derivatives)	double[][]	X, Y, X1, ...
Element matrix	double[][]	A

LISTING 1: Local assembly source code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange $p = 1$ elements.

```
void helmholtz(double A[3][3], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W[3] = {...}
  static const double X_D10[3][3] = {{...}}
  static const double X_D01[3][3] = {{...}}

  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      for (int k = 0; k<3; k++)
        A[j][k] += ((Y[i][k]*Y[i][j]+
          +((K1*X_D10[i][k]+K3*X_D01[i][k])*(K1*X_D10[i][j]+K3*X_D01[i][j]))+
          +((K0*X_D10[i][k]+K2*X_D01[i][k])*(K0*X_D10[i][j]+K2*X_D01[i][j])))*
          *det*W[i]);
}
```

discretized domain and coefficient fields, for instance indicating the values of velocity or pressure in 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 2D 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 I shows the variable names that 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 of 3 to 15), and the computation reduces to a summation of a few products involving basis functions. An example is provided in Listing 1, which shows the assembly kernel for a Helmholtz problem using Lagrange basis functions on 2D elements with polynomial order $p = 1$. In other scenarios, such as when solving the Burgers equation, the number of arrays involved in the computation of the element matrix can be much larger. The assembly source code is given in Listing 2 and contains 14 different arrays that are accessed, where the same array can be referenced multiple

LISTING 2: Local assembly source code generated by Firedrake for a Burgers problem on a 3D tetrahedral mesh using Lagrange $p = 1$ elements.

```
void burgers(double A[12][12], double **coords, double **w) {
 // K, det = Compute Jacobian (coords)

 static const double W[5] = {...}
 static const double X1_D001[5][12] = {{...}}
 static const double X2_D001[5][12] = {{...}}
 //11 other basis functions definitions.
 ...
 for (int i = 0; i<5; i++) {
  double F0 = 0.0;
  //10 other declarations (F1, F2,...)
  ...
  for (int r = 0; r<12; r++) {
   F0 += (w[r][0]*X1_D100[i][r]);
   //10 analogous statements (F1, F2, ...)
   ...
  }
  for (int j = 0; j<12; j++)
   for (int k = 0; k<12; k++)
    A[j][k] += (..(K5*F9)+(K8*F10))*Y1[i][j])+
    +(((K0*X1_D100[i][k])+(K3*X1_D010[i][k])+(K6*X1_D001[i][k]))*Y2[i][j]))*F11)+
    +(..((K2*X2_D100[i][k]) + ··· + (K8*X2_D001[i][k]))*((K2*X2_D100[i][j]) + ··· +
    (K8*X2_D001[i][j]))..)+
    + <roughly a hundred sum/muls go here>)..)*
    *det*W[i]);
 }
}
```

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 proportionately to the order of the method, and arrays may be block sparse. In addition to a larger number of operations, more complex cases like the Burgers equation are characterized by high register pressure.

The Helmholtz and Burgers equations exemplify a large class of problems and, as such, will constitute our benchmark problems, along with the Diffusion equation. We carefully motivate this choice in Section 5.2.1.

Despite the infinite variety of assembly kernels that Firedrake can generate, it is still possible to identify common domain-specific traits that can be exploited for effective code transformations and SIMD vectorization. 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 precomputation of values (e.g., the F values in Burgers) and introduction of temporary arrays, as explained in Section 3; (3) depending on the problem being solved, the `j` and `k` loops could iterate over the same iteration space; and (4) most of the subexpressions on the right-hand side of the element matrix computation depend on just two loops (either `i-j` or `i-k`). In Section 3, we show how to exploit these observations to define a set of systematic, composable optimizations.

3. CODE TRANSFORMATIONS

The code transformations presented in this section are applicable to all finite element problems that can be formulated in Firedrake. Their generality and the potential for applicability in other domains is discussed in Section 6.

As already emphasized, the variations in the structure of mathematical expressions and in loop trip counts, although typically limited to tens of iterations, render the optimization process challenging, requiring distinct sets of transformations to bring performance closest to the machine peak in different problems. For example, the Burgers problem in Listing 2, given the large number of arrays accessed, suffers from high register pressure, whereas the Helmholtz problem in Listing 1 does not; this 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 class of problems and architectures.

3.1. Padding and Data Alignment

The absence of stencils renders the element matrix computation easily autovectorizable by a vendor compiler. Nevertheless, autovectorization 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. First, all arrays are allocated to addresses that are multiples of `VL`. Then, 2D arrays are padded by rounding the number of columns to the nearest multiple of `VL`. For instance, assume that the original size of a basis function array is 3×3 and `VL` $= 4$ (e.g., AVX processor, with 256-bit-long vector registers and 64-bit 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 2D 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.

3.2. Generalized Loop-Invariant Code Motion

From an inspection of the codes in Listings 1 and 2, it can be noted that the computation of A involves evaluating many subexpressions that only depend on two iteration variables. Since symbols in most of these subexpressions are read-only variables, there is ample room for loop-invariant code motion. Vendor compilers apply this technique, although not in the systematic way we need for our assembly kernels. We want to overcome two deficiencies that both Intel and GNU compilers exhibit. First, they only identify subexpressions that are invariant with respect to the innermost loop. This is an issue for subexpressions depending on `i-k`, which are not automatically lifted in the loop order `ijk`. Second, the hoisted code is scalar and therefore not subjected to autovectorization.

We work around these limitations with source-level loop-invariant code motion. In particular, we precompute all values that an invariant subexpression assumes along its fastest varying dimension. This is implemented by introducing a temporary array per invariant subexpression 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 autovectorized as

LISTING 3: Local assembly source code for the Helmholtz problem in Listing 1 after application of padding, data alignment, and loop-invariant code motion, for an AVX architecture. In this example, subexpressions invariant to `j` are identical to those invariant to `k`, so they can be precomputed once in the *r* loop.

```
void helmholtz(double A[3][4], double **coords) {
 #define ALIGN __attribute__((aligned(32)))
 // K, det = Compute Jacobian (coords)

 static const double W[3] ALIGN = {...}
 static const double X_D10[3][4] ALIGN = {{...}}
 static const double X_D01[3][4] ALIGN = {{...}}

 for (int i = 0; i<3; i++) {
  double LI_0[4] ALIGN;
  double LI_1[4] ALIGN;
  for (int r = 0; r<4; r++) {
   LI_0[r] = ((K1*X_D10[i][r])+(K3*X_D01[i][r]));
   LI_1[r] = ((K0*X_D10[i][r])+(K2*X_D01[i][r]));
  }
  for (int j = 0; j<3; j++)
   #pragma vector aligned
   for (int k = 0; k<4; k++)
    A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j]+LI_1[k]*LI_1[j])*det*W[i]);
 }
}
```

part of an inner loop. Given the short trip counts of our loops, it is important to achieve autovectorization of hoisted terms to minimize the percentage of scalar instructions, which could otherwise be significant. It is also worth noting that in some problems, for instance Helmholtz, invariant subexpressions along `j` are identical to those along `k`, and both loops iterate over the same iteration space, as anticipated in Section 2. In these cases, we safely avoid redundant precomputation.

Listing 3 shows the Helmholtz assembly source code after the application of loop-invariant code motion, padding, and data alignment.

3.3. 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. In COFFEE, these optimizations are supported either by means of explicit code transformations (interchange, unroll-and-jam) or indirectly by delegation to the compiler through standard pragmas (unroll). Tiling at the level of vector registers is an additional feature of COFFEE. 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 code snippet in Listing 3 and ignore the presence of the operation `det*W[i]`, the computation of the element matrix is abstractly expressible as

$$A_{jk} = \sum_{\substack{x \in B' \subseteq B \\ y \in B'' \subseteq B}} x_j \cdot y_k \quad j, k = 0, \ldots, 2, \tag{1}$$

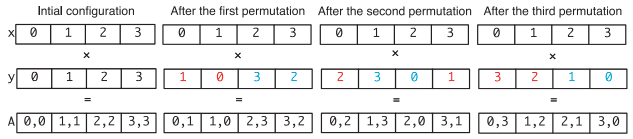

Intial configuration
After the first permutation
After the second permutation
After the third permutation
x
y
A
×
=

Fig. 2. Outer-product vectorization by permuting values in a vector register.

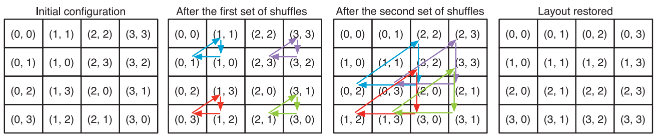

Initial configuration
After the first set of shuffles
After the second set of shuffles
Layout restored

Fig. 3. 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.

LISTING 4: Local assembly source code generated by Firedrake for the Helmholtz problem after application of *op-vect* on top of the optimizations shown in Listing 3. In this example, the unroll-and-jam factor is 1.

```
void helmholtz(double A[4][4], double **coords) {
  // K, det = Compute Jacobian (coords)
  // Declaration of basis function matrices

  for (int i = 0; i<3; i++) {
    // Do generalized loop-invariant code motion
    for (int j = 0; j<4; j+=4)
      for (int k = 0; k<4; k+=4) {
        // load and set intrinsics
        // Compute A[0,0], A[1,1], A[2,2], A[3,3]
        // One permute_pd intrinsic per k-loop load
        // Compute A[0,1], A[1,0], A[2,3], A[3,2]
        // One permute2f128_pd intrinsic per k-loop load
        // . . .
      }
    // Remainder loop (from j = 4 to j = 6)
  }
  // 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<4; 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 A[j+1][k], A[j+2][k], A[j+3][k]
    }
  }
}
```

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 2 illustrates how we can evaluate 16 entries ($j, k = 0, \ldots, 3$) of the element matrix using just two 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 autovectorization for both GNU and Intel compilers, instead, executes four 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, executed once outside of the `ijk` loop nest. The generated pseudocode for the simple Helmholtz problem when using *op-vect* is shown in Listing 4.

3.4. Expression Splitting

In complex kernels, like Burgers in Listing 2, and on certain architectures, achieving effective register allocation can be challenging. If the number of variables independent of the innermost-loop dimension is close to or greater than the number of available CPU registers, then 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 subexpressions; an example for the Helmholtz problem is given later in Listing 5. The transformation can be regarded as a special case of classic loop fission, in which associativity of the sum is exploited to distribute the expression across multiple loops. To the best of our knowledge, expression splitting is not supported by available compilers.

LISTING 5: Local assembly source code generated by Firedrake for the Helmholtz problem in which *split* has been applied on top of the optimizations shown in Listing 3. In this example, the split factor is 2.

```
void helmholtz(double A[3][4], double **coords) {
 // Same code as in Listing~3 up to the j loop
 for (int j = 0; j<3; j++)
  #pragma vector aligned
  for (int k = 0; k<4; k++)
   A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j])*det*W[i];
 for (int j = 0; j<3; j++)
  #pragma vector aligned
 for (int k = 0; k<4; k++)
   A[j][k] += LI_1[k]*LI_1[j]*det*W[i];
}
```

Splitting an expression (henceforth *split*) has several drawbacks, however. First, it increases the number of accesses to A in proportion to the "split factor," which is the number of subexpressions produced. In addition, depending on how splitting is done, it can lead to redundant computation. For example, the number of times the product $det * \texttt{W}[i]$ is performed is proportional to the number of subexpressions, as shown in the code snippet. Further, it increases loop overhead, such as through additional branch instructions. Finally, it might affect register locality: for instance, the same array could be accessed in different subexpressions, requiring a proportional number of loads be performed, although this is not the case for the Helmholtz example. Nevertheless, the performance gain from improved register reuse can still be greater if suitable heuristics are used. Our approach consists of traversing the expression tree and recursively splitting it into multiple subexpressions as long as the number of variables independent of the innermost loop exceeds a certain threshold. This is elaborated in the next section and validated against empirical search in Section 5.2.6.

4. OVERVIEW OF COFFEE

Firedrake users employ the UFL to express problems in a notation resembling mathematical equations. At runtime, the high-level specification is translated by a modified version of the FFC [Kirby and Logg 2006] into an AST representation of one or

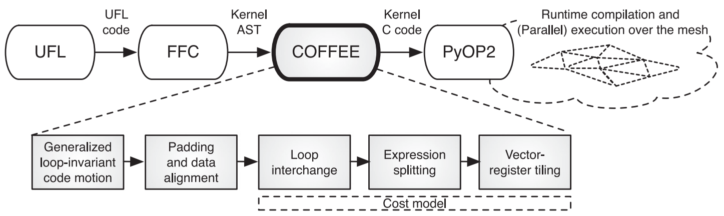

UFL
UFL code
FFC
Kernel AST
COFFEE
Kernel C code
PyOP2
Runtime compilation and (Parallel) execution over the mesh
Generalized loop-invariant code motion
Padding and data alignment
Loop interchange
Expression splitting
Vector-register tiling
Cost model

Fig. 4. High-level view of Firedrake. COFFEE is at the core, receiving ASTs from a modified version of the FFC and producing optimized C code kernels.

more finite element assembly kernels. ASTs are then passed to COFFEE to apply the transformations described in Section 3. The output of COFFEE, C code, is eventually provided to PyOP2 [Rathgeber et al. 2012; 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 4.

Some of the transformations presented in the previous section, as well as other typical compiler optimizations used in COFFEE, such as loop interchange and unrolling, are parametric; for instance, expression splitting and loop unrolling depend on the split and the unroll factors, respectively. Since all transformations are composable, the optimization space can be as large as thousands of variants for local assembly kernels in which mathematical expressions are moderately complex and the iteration space is not particularly small; so, a pruning strategy is required. For this, COFFEE uses heuristics and a simple cost model to select the most suitable optimization strategy for a given problem. Autotuning can also be used, although it would significantly increase the runtime overhead, since the generation of ASTs occurs at runtime as soon as problem-specific data are available.

The compiler applies an ordered sequence of optimization steps to the ASTs received from FFC. Application of loop-invariant code motion (or *licm*) must precede padding and data alignment, due to the introduction of temporary arrays. These transformations are always performed because they are likely to improve the runtime performance, as demonstrated by our results in Section 5.2. Based on a cost model (described later), loop interchange, *split*, and *op-vect* may be introduced. Their implementation is centered on analysis and manipulation of the kernel AST. When *op-vect* is selected, the compiler outputs AVX or AVX-512 intrinsics code. Any possible corner cases are handled: for example, if *op-vect* 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 autovectorization, is inserted.

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. Our compiler chooses the outermost loop such that the number of invariant loads is smallest, freeing additional registers for computation of the element matrix.

Loop unrolling. 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.

```
1  Input: n_outer_arrays, n_inner_arrays, n_consts, n_regs
2  Output: uaj_factor, split_factor
3  n_outer_regs = n_regs / 2
4  split_factor = 0
5  // Compute splting factor
6  while n_outer_arrays > n_outer_regs
7    n_outer_arrays = n_outer_arrays / 2
8    split_factor = split_factor + 1
9  // Compute unroll-and-jam factor for op-vect
10 n_regs_avail = n_regs - (n_outer_arrays + n_consts)
11 uaj_factor = n_reg_avail / n_inner_arrays
12 // Estimate the benefit of permuting loops
13 permute = n_outer_arrays > n_inner_arrays
14 return <permute, split_factor, uaj_factor>
```

Fig. 5. The cost model is employed by the compiler to estimate the most suitable unroll-and-jam (when *op-vect* is used) and split factors, avoiding the overhead of autotuning.

We first observe that manual full (or extensive) unrolling is unlikely to be effective for two reasons. First, the `ijk` loop nest would need to be small enough such that the unrolled instructions do not exceed the instruction cache, which is rarely the case: for the local assembly kernels we study, the minimum size of the `ijk` loop nest is $3 \times 3 \times 3$ (triangular mesh and polynomial order 1); however, this increases rapidly with the polynomial order of the method and problem dimensionality (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 quite common in practice. Second, manual unrolling is dangerous because it may compromise compiler autovectorization 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 note that recent versions of compilers such as those of GNU and Intel estimate close-to-optimal unroll factors when the loops are affine and their bounds are relatively small and known at compile time, which is the case of our kernels. Our choice, therefore, is to leave the backend compiler in charge of selecting unroll factors. This also simplifies COFFEE's cost model, described next. The only situation in which we explicitly unroll-and-jam a loop is when *op-vect* is used, since the transformed code prevents the Intel compiler from applying this optimization, even if specific pragmas are added.

Cost model. The cost model is shown in Figure 5. It takes into account the number of available logical vector registers, `n_regs`, and the number of unique variables accessed: `n_consts` counts variables independent of both `j` and `k` loops and temporary registers, `n_outer_arrays` counts `j`-dependent variables, and `n_inner_arrays` counts `k`-dependent variables, assuming the `ijk` loop order. These values are used to estimate unroll-and-jam and split factors for *op-vect* and *split*. If a factor is 0, then the corresponding transformation is not applied. The *split* transformation is triggered whenever the number of hoistable terms is larger than the available registers along the outer dimension (lines 3 through 8), which is approximated as half of the total (line 3). A split factor of n means that the assembly expression should be "cut" into n subexpressions. Depending on the structure of the assembly expression, each subexpression might end up accessing a different number of arrays; the cost model is simplified by assuming that all subexpressions are of the same size. The unroll-and-jam factor for the *op-vect* transformation is determined as a function of the available logical registers—that is, those not used for storing hoisted terms (lines 9 through 11). Finally, the profitability of loop interchange is evaluated (line 13).

5. PERFORMANCE EVALUATION

5.1. Experimental Setup

Experiments were run on a single core of two Intel architectures: a Sandy Bridge (I7-2600 CPU, running at 3.4GHz, 32KB L1 cache, and 256KB L2 cache) and a Xeon Phi (5110P, running at 1.05GHz in native mode, 32KB L1 cache, and 512KB L2 cache). We have chosen these two architectures because of the differences in the 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 `icc 13.1` compiler was used. We selected the best optimization levels for each platform; we used `-xAVX` for autovectorization and `-O2` on the Sandy Bridge, and `-O3` on the Xeon Phi.

We present two studies. First, in Section 5.2, we analyze the impact of the proposed code transformations, and their interplay, in three real-world representative equations. We quantify the speedups achievable at the level of the local assembly kernel, by excluding the other stages of the computation from the measurements. Then, in Section 5.3, we provide a full application investigation. Here, we use another equation, bringing the total number of examined problems to four, to demonstrate that COFFEE's optimizations actually allow a significant reduction in the overall execution time, which is the motivation for this work.

5.2. In-Depth Performance Analysis of the Code Transformations and Their Interplay

5.2.1. Choice and Properties of the Benchmarks. Our code transformations were evaluated in three real-world problems based on the following PDEs: (1) Helmholtz, (2) Advection-Diffusion, and (3) Burgers.

The three chosen benchmarks 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 optimizations 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 because it applies to a vector-valued quantity, wherease the elliptic operators apply to scalar quantities; this impacts the generated code, as explained next. The operators that 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 in Luporini [2014a]) and executed over real unstructured meshes through Firedrake. The Helmholtz code has already been shown in Listing 1. For Advection-Diffusion, the Diffusion equation, which uses the same differential operators as Helmholtz, is considered. 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 nonlinear problem employing differential operators different from those of Helmholtz and relying on vector-valued quantities, which has a major impact on

Table II. Performance Improvement Due to Generalized Loop-Invariant Code Motion over the Original Nonoptimized Code

		Sandy Bridge				Xeon Phi			
problem	shape	p1	p2	p3	p4	p1	p2	p3	p4
Helmholtz	triangle	1.05	1.46	1.68	1.67	1.49	1.06	1.05	1.17
Helmholtz	tetrahedron	1.36	2.10	2.64	2.27	1.28	1.29	2.05	1.73
Helmholtz	prism	2.16	2.28	2.45	2.06	1.04	2.26	1.93	1.64
Diffusion	triangle	1.09	1.68	1.97	1.64	1.07	1.06	1.18	1.16
Diffusion	tetrahedron	1.30	2.20	3.12	2.60	1.00	1.38	2.02	1.74
Diffusion	prism	2.15	1.82	2.71	2.32	1.11	2.16	1.85	2.83
Burgers	triangle	1.53	1.81	2.68	2.46	1.21	1.42	2.34	2.97
Burgers	tetrahedron	1.61	2.24	1.69	1.59	1.01	2.55	0.98	1.21
Burgers	prism	2.11	2.20	1.66	1.32	1.39	1.56	1.18	1.04

the generated assembly source code (see Listing 2), where a larger number of basis function arrays ($X1, X2, \ldots$) and constants ($F0, F1, \ldots, K0, K1, \ldots$) are generated.

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

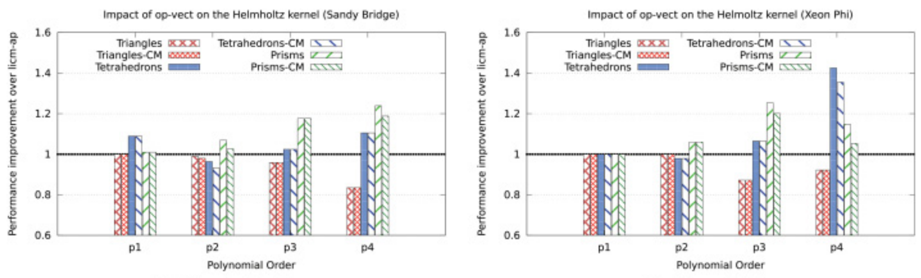
5.2.2. Loop Interchange. In the following, only results for the loop order `ijk` are shown. For the considerations exposed in Section 4, loop interchanges having an inner loop along `i` caused slowdowns; additionally, interchanging `j` and `k` loops while keeping `i` as outermost loop did not provide any benefits.

5.2.3. Impact of Generalized Loop-Invariant Code Motion. Table II illustrates the performance improvement obtained when loop-invariant code motion (or *licm*) is applied. In general, the speedups are notable. The main reasons were anticipated in Section 3.2: in the original code, (1) subexpressions invariant to outer loops are not automatically hoisted, whereas (2) subexpressions invariant to the innermost loop are hoisted but their execution is not autovectorized. 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., higher dimensional elements 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 autovectorization. For instance, on the Sandy Bridge, the improvement at $p = 3$ is larger than that at $p = 4$ 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. The remainder loop overhead is more pronounced on the Xeon Phi,

Table III. Performance Improvement Due to Generalized Loop-Invariant Code Motion, Data Alignment, and Padding over the Original Nonoptimized Code

		Sandy Bridge				Xeon Phi			
problem	shape	p1	p2	p3	p4	p1	p2	p3	p4
Helmholtz	triangle	1.32	1.88	2.87	4.13	1.50	2.41	1.30	1.96
Helmholtz	tetrahedron	1.35	3.32	2.66	3.27	1.41	1.50	2.79	2.81
Helmholtz	prism	2.63	2.74	2.43	2.75	2.38	2.47	2.15	1.71
Diffusion	triangle	1.38	1.99	3.07	4.28	1.08	1.88	1.20	1.97
Diffusion	tetrahedron	1.41	3.70	3.18	3.82	1.05	1.51	2.76	3.00
Diffusion	prism	2.55	3.13	2.73	2.69	2.41	2.52	2.05	2.48
Burgers	triangle	1.56	2.28	2.61	2.77	2.84	2.26	3.96	4.27
Burgers	tetrahedron	1.61	2.10	1.60	1.78	1.48	3.83	1.55	1.29
Burgers	prism	2.19	2.32	1.64	1.42	2.18	2.82	1.24	1.25



(a) Sandy Bridge (b) Xeon Phi

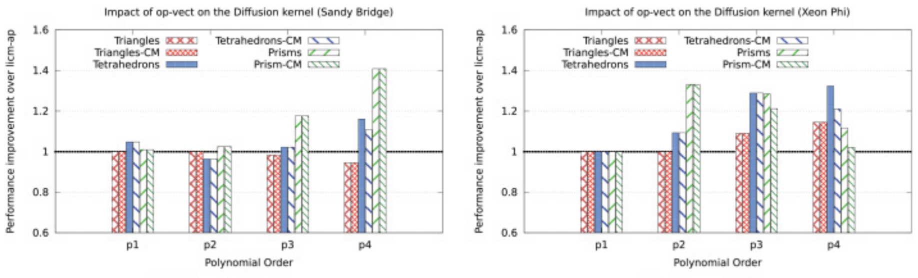
Fig. 6. Performance improvement over *licm-ap* obtained by *op-vect* in the Helmholtz kernel. Bars suffixed with "CM" indicate that the cost model was used to transform the kernel.

where the vector length is twice as long, which leads to proportionately larger scalar remainder loops.

5.2.4. Impact of Padding and Data Alignment. Table III shows the cumulative impact of loop-invariant code motion (or *licm*), data alignment, and padding over the original code. In the following, this version of the code is referred to as *licm-ap*. Padding, which avoids the introduction of a remainder loop as described in Section 5.2.3, as well as data alignment, enhance the quality of autovectorization. Occasionally, the execution time of *licm-ap* is close to that of *licm*, since the nonpadded element matrix size is already a multiple of the vector length. Rarely, *licm-ap* is slower than *licm* (e.g., in Burgers $p = 3$ on the Sandy Bridge). One possible explanation is that the number of aligned temporaries introduced by *licm* is so large to induce cache associativity conflicts.

5.2.5. Impact of Vector-Register Tiling. In this section, we evaluate the impact of vector-register tiling. We compare two versions: the baseline, *licm-ap*; and vector-register tiling on top of *licm-ap*, which in the following is referred to simply as *op-vect*.

Figures 6 and 7 illustrate the speedup achieved by *op-vect* over *licm-ap* in the Helmoltz and Diffusion kernels, respectively. As explained in Section 4, the *op-vect* version requires the unroll-and-jam factor to be explicitly set. To distinguish between the two ways in which this parameter was determined, for each problem instance (equation, element shape, polynomial order) we report two values: the best speedup obtained after all feasible unroll-and-jam factors were tried and the speedup when the unroll-and-jam factor was retrieved via the cost model. In a plot legend, cost model bars are suffixed with "CM."



(a) Sandy Bridge (b) Xeon Phi

Fig. 7. Performance improvement over *licm-ap* obtained by *op-vect* in the Diffusion kernel. Bars suffixed with “CM” indicate that the cost model was used to transform the kernel.

It is worth noticing that in most cases, the cost model successfully determines how to transform a kernel to maximize its performance. This is chiefly because assembly kernels fit the L1 cache; thus, within a certain degree of confidence, it is possible to predict how to obtain a fast implementation by simply reasoning on the register pressure. For each problem, the cost model stated to use the default loop permutation, to apply a particular unroll-and-jam factor, and not to perform expression splitting, which, as explained in Section 5.2.6, only deteriorates performance in Helmholtz and Diffusion.

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 reuse and, therefore, to maximize the performance. However, as the iteration space becomes larger, *op-vect* leads to improvements up to 1.4× on the Sandy Bridge (Diffusion, prismatic mesh, $p = 4$—increasing the overall speed up from 2.69× to 3.87×), and up to 1.4× on the Xeon Phi (Helmholtz, tetrahedral mesh, $p = 3$—bringing the overall speed up from 1.71× to 2.42×).

Using the Intel Architecture Code Analyzer tool [Intel Corporation 2012] on the Sandy Bridge, we confirmed that speedups are a consequence of increased register reuse. In Helmholtz $p = 4$, 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.

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

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

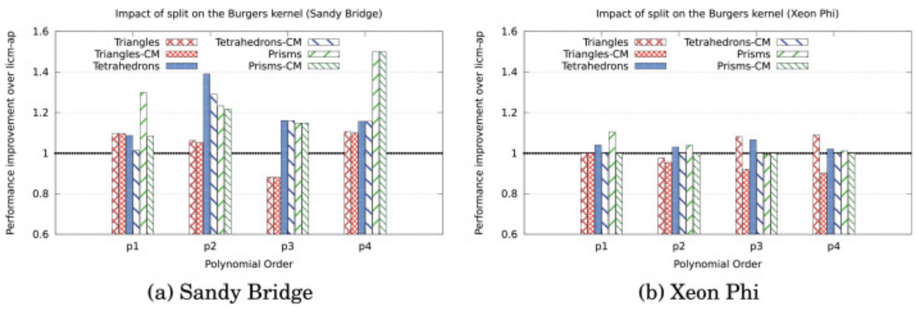


Fig. 8. Performance improvement over *licm-ap* obtained by *split* in the Burgers kernel. Bars suffixed with “CM” indicate that the cost model was used to transform the kernel.

5.2.6. 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 3.4, the price to pay for this optimization is an increased number of accesses to the element matrix and, potentially, redundant computation. Similarly to the analysis of vector-register tiling, we compare two versions: the baseline, *licm-ap*, and expression splitting on top of *licm-ap*, which, for simplicity, in the following is referred to as *split*.

For the Helmholtz and Diffusion kernels, in which only between four and eight temporaries are read at every loop iteration, `split` tends to slow down the computation because of the aforementioned drawbacks. Slowdowns up to 1.4× and up to 1.6× were observed, respectively, on the Sandy Bridge and the Xeon Phi. Note that the cost model prevents the adoption of the transformation: the `while` statement in Figure 5 is never entered.

In the Burgers kernels, between 12 and 24 temporaries are accessed at every loop iteration, so *split* plays a key role on the Sandy Bridge, where the number of available logical registers is only 16. Figure 8 shows the performance improvement achieved by *split* over *licm-ap*. In almost all cases, a split factor of 1, meaning that the original expression was divided into two parts, ensured close-to-peak performance. The transformation negligibly affected register locality, so speedups up to 1.5× were observed. For instance, on the Sandy Bridge, when $p = 4$ and a prismatic mesh is employed, the overall performance improvement (i.e., the one over the original code) increases from 1.44× to 2.11×. On the Xeon Phi, the impact of *split* is only marginal, since register spilling is limited by the presence of 32 logical vector units.

On the Sandy Bridge, the performance of the Burgers kernel on a prismatic mesh was 20.0GFlop/s from $p = 1$ to $p = 3$, whereas it was 21.3GFlop/s in the case of $p = 4$. These values are notably close to the peak performance of 30.4GFlop/s. Disabling *split* makes the performance drop to 17.0GFlop/s for $p = 1, 2$, 18.2GFlop/s for $p = 3$, and 14.3GFlop/s for $p = 4$. These values are in line with the speedups shown in Figure 8.

The *split* transformation was also tried in combination with *op-vect* (*split-op-vect*), although the cost model prevents its adoption on both platforms. Despite improvements up to 1.22×, *split-op-vect* never outperforms *split*. This is motivated by two factors: for small split factors, such as 1 and 2, the data space to be tiled is still too big, and register spilling affects runtime; for higher ones, subexpressions become so small that, as explained in Section 5.2.5, extensive unrolling already ensures high register reuse.

5.2.7. Comparison with FFC’s Built-In Optimizations. We have modified the FFC to return an AST representation of a local assembly kernel rather than plain C++ code to enable

Table IV. Performance Comparison between FEniCS and COFFEE on Sandy Bridge

Problem	Max Slowdown	Max Speedup
Helmholtz	—	4.14×
Diffusion	—	4.28×
Burgers	2.24×	1.61×

code transformations in COFFEE. In FEniCS, FFC applies its own optimizations to local assembly kernels [Ølgaard and Wells 2010], which mainly consist of loop-invariant code motion and elimination, of floating point operations involving zero-valued entries in basis function arrays.

FFC's loop-invariant code motion is different from that of COFFEE. It is based on expansion of arithmetic operations, such as applying distributivity and associativity to products and sums at code generation time, to identify terms that are invariant of the whole loop nest. Depending on the way expansion is performed, operation count may not decrease significantly.

Elimination of zero-valued terms, which are the result of using vector-valued quantities in the finite element problem, has the effect of introducing indirection arrays in the generated code. This kind of optimization is currently under development in COFFEE, although it will differ from that of FEniCS by avoiding noncontiguous memory accesses, which would otherwise affect vectorization, at the price of removing fewer zero-valued contributions.

Table IV summarizes the performance achieved by COFFEE over the *fastest* FEniCS (FFC) implementation on the Sandy Bridge for the Burgers, Helmholtz, and Diffusion kernels. Burgers' slowdowns occur in presence of a small iteration space (triangular mesh, $p \in [1, 2]$; tetrahedral mesh, $p \in [1, 2]$; prismatic mesh, $p = 1$). The result shown represents the worst slowdown, which was obtained with a triangular mesh and $p = 1$. This is a result of removing zero-valued entries in FEniCS's basis function arrays: some operations are avoided, but indirection arrays prevent autovectorization, which significantly impacts performance as soon as the element matrix becomes bigger than 12×12. However, with the forthcoming zero-removal optimization in COFFEE, we expect to outperform FEniCS in all problems. In the cases of Helmholtz and Diffusion, the minimum improvements, respectively, are 1.10× and 1.18× (2D mesh, $p = 1$), which tend to increase with polynomial order and element dimensionality up to the values illustrated in Table IV.

5.2.8. Comparison with Handmade BLAS-Based Implementations. For the Helmholtz problem on a tetrahedral mesh, manual implementations based on `Intel MKL BLAS` were tested on the Sandy Bridge. This particular kernel can be easily reduced to a sequence of four matrix-matrix multiplies that can be computed via calls to BLAS `dgemm`. In the case of $p = 4$, where the element matrix is of size 35×35, the computation was almost twice as slow as the case in which *licm-ap* was used, with the slowdown being even worse for smaller problem sizes. These experiments suggest that the question regarding to what extent linear algebra libraries can improve performance cannot be trivially answered. This is due to a combination of issues: the potential loss in data locality, as exposed in Section 3.4; the actual effectiveness of these libraries when the arrays are relatively small; and the problem inherent to assembly kernels concerning extraction of matrix-matrix multiplies from static analysis of the kernel's code. A comprehensive study of these aspects will be addressed in further work.

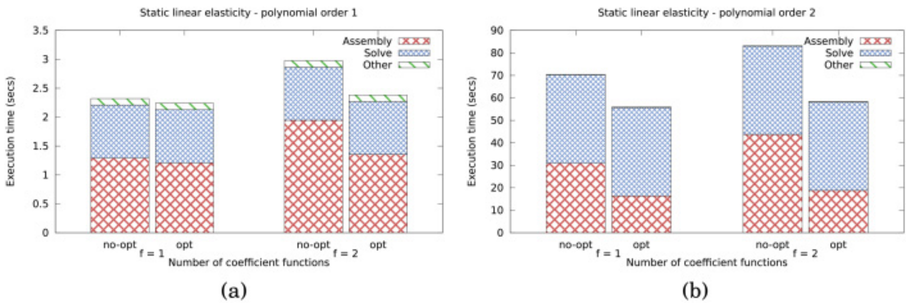


Fig. 9. Performance improvement over nonoptimized code for the static linear elasticity equation on a single core of the Sandy Bridge architecture.

5.2.9. On the Compilation Time. Firedrake and its various software components (see Figure 4) are implemented in Python and Cython, whereas the generated code is pure C. COFFEE has been written in Python to be naturally integrated within Firedrake. Among the various Firedrake modules, COFFEE is where, in general, the least amount of time is spent. The process of transforming the AST usually takes order of milliseconds. For instance, for the Helmholtz equation with $p = 4$ on a tetrahedral mesh, it took 0.9ms to transform the AST, whereas the whole assembly process needed 3.84ms. Being a domain-specific compiler, COFFEE expects C code with the structure outlined in Figure 1, so typical compilers functionalities such as dependency tracking are simplified with respect to a general-purpose compiler, which implies a clear advantage in compilation time and implementation complexity.

5.3. Full Application Study

In this section, we investigate the performance gain for an entire finite element computation developed in Firedrake, a static linear elasticity problem. The source code is available in 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 in a “global” sparse matrix and a global vector, respectively, 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, such as in Ølgaard and Wells [2010], the computational cost of local assembly grows with the *complexity* of the PDE (e.g., because of larger loops and more articulated expressions), whereas the solve time increases with polynomial order and mesh discretization. The complexity of an equation depends on several 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 $p = 1$ and $p = 2$. To run our experiments, we use a single core of the Sandy Bridge architecture described in Section 5. The application runs on a tetrahedral mesh composed of 196,608 elements.

In Figure 9, 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`—e.g., 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 transformations, only a minimal performance gain is achieved. However, if either p 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 that 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 (see Section 3.2).

In these experiments, we observe a maximum performance improvement of $1.47\times$ over the nonoptimized local assembly source code, obtained in the case ($f = 2, p = 2$). However, we reiterate the fact that full application speedups increase proportionately 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 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.

6. 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 than 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 UFL and the FFC, 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 DSL 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 AST, in the case of Firedrake) 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 that 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:

$$\forall_{i,j} \quad A_{ij}^{K} = \sum_{q=1}^{n_1} \sum_{k=1}^{n_2} \alpha_{k,q}(a', b', c', \ldots)\beta_{q,i,j}(a, b, c, d, \ldots)\gamma_q(w_K, z_K). \tag{2}$$

The expression represents the numerical evaluation of an integral at n_1 points in the mesh element K computing the local element matrix A. Functions α, β, and γ are problem specific and can be intricately complex, involving, for example, the evaluation of derivatives. However, we can 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 subexpressions) 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 reschedule the computation suitably for the underlying architecture. Loop-invariant code motion and expression splitting follow this principle, so they can be readapted 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 (2), the summations correspond to reduction loops, whereas loops over indices i and j are fully parallel. Throughout the article, we assumed that a kernel will be executed by a single thread, which is likely to be the best strategy for standard multicore CPUs. On the other hand, we note that for certain architectures (e.g., GPUs), this could be prohibitive due to memory requirements. Intrakernel 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 that this is the right approach to achieve performance portability.

(2) The various subexpressions in β only depend on (i.e., iterate along) a subset of the enclosing loops. In addition, subexpressions may be computed redundantly in different iteration spaces. This code structure motivated the generalized loop-invariant code motion technique. The intuition is that whenever subexpressions 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. Precomputation 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 rescheduling (reassociating) 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, with each part characterized by less stringent memory requirements (a variant of this idea for nonaffine loops in unstructured mesh applications has been adopted in Bertolli et al. [2013]). In Equation (2), for instance, not only can any of the functions α, β, and γ be split (assuming that 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 that 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, such as n_1 and n_2 in Equation (2), are known at code generation time. This means that "good" code can be generated. For example, loop bounds can be made explicit, arrays can be statically initialized, and pointer aliasing is easily avoidable. Further, all of these factors contribute to the applicability and the effectiveness of some of our code transformations. For instance, knowing loop bounds allows both generation of correct code when applying vector-register tiling (see Section 3.3) and discovery of redundant computation opportunities (see Section 3.2). Padding and data alignment are special cases, as they could be performed at runtime 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 dataflow analysis is performed to ensure that the extra loop iterations over the padded region do not affect the numerical results.

—*Multiloop vectorization*: Compiler autovectorization has become increasingly effective in a variety of codes. However, to the best of our knowledge, multiloop 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 article shows that two-loop vectorization can outperform standard autovectorization. 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 multiloop 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 multiloop vectorization in a real scenario, our research represents an incentive for studying this technique in a broader and systematic way.

7. RELATED WORK

The finite element method is extensively used to approximate solutions of PDEs. Well-known frameworks and applications include Nek5000 [Fischer et al. 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 DSLs to decouple the finite element specification from its underlying implementation has facilitated 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 [Ølgaard and Wells 2010], which motivates our work in COFFEE.

Optimization of quadrature-based local assembly for CPU architectures has been addressed in FEniCS [Ølgaard and Wells 2010]. The comparison between COFFEE and this work has been presented in Section 5.2.7. 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 Section 3 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 subexpressions 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 readapted 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 autogenerating optimized code for other domains: Spiral [Püschel et al. 2005] for digital signal processing numerical algorithms, LGen [Spampinato and Püschel 2014] for dense linear algebra, and 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, such as 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 autotuning 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 reduce complexity and overhead. Likewise, COFFEE uses a cost model and heuristics (Section 4) to steer the optimization process.

8. CONCLUSIONS

In this article, 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, as well as 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 quite small, with the size depending on aspects like the 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 proportionately 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 simple-minded 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 two state-of-the-art Intel platforms. Further work includes a comprehensive study about feasibility and constraints on transforming kernels into a sequence of calls to external linear algebra libraries. As suggested by the performance evaluation, an exploration of more sophisticated approaches for expression splitting also looks promising. COFFEE is already integrated with Firedrake and supports all of the problems expressible in Firedrake. In addition, we have discussed the generality and applicability of the proposed code transformations to other domains.

ACKNOWLEDGMENTS

The authors would like to thank Carlo Bertolli, Lawrence Mitchell and Francis Russell for their invaluable suggestions and their contribution to the Firedrake project. We thank the ACM *TACO* and HiPEAC '15 reviewers for their suggestions that led to a significantly improved version.

REFERENCES

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman (Eds.). 2007. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson/Addison Wesley, Boston, MA. http://www.loc.gov/catdir/toc/ecip0618/2006024333.html.

M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. 2014. Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software* 40, 2, Article No. 9. DOI:http://dx.doi.org/10.1145/2566630.

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

C. Bertolli, A. Betts, N. Loriant, G. R. Mudalige, D. Radford, D. A. Ham, M. B. Giles, and P. H. J. Kelly. 2013. Compiler optimizations for industrial unstructured mesh CFD applications on GPUs. In *Languages and Compilers for Parallel Computing*, Hironori Kasahara and Keiji Kimura (Eds.). Lecture Notes in Computer Science, Vol. 7760. Springer, 112–126. DOI:http://dx.doi.org/10.1007/978-3-642-37658-0_8

U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 101–113. DOI:http://dx.doi.org/10.1145/1375581.1375595

Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. 2011. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. ACM, New York, NY, Article No. 9. DOI:http://dx.doi.org/10.1145/2063384.2063396

Firedrake. 2014. The Firedrake Project. Retrieved November 16, 2014, from http://www.firedrakeproject.org.

P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier. 2008. Nek5000 Web Page. Retrieved November 16, 2014, from http://nek5000.mcs.anl.gov.

M. Frigo and S. G. Johnson. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE* 93, 2, 216–231.

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

T. Henretty, R. Veras, F. Franchetti, L.-.N Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A stencil compiler for short-vector SIMD architectures. In *Proceedings of the 27th International ACM Conference on Supercomputing (ICS'13)*. ACM, New York, NY, 13–24. DOI:http://dx.doi.org/10.1145/2464996.2467268

Intel Corporation. 2012. *Intel Architecture Code Analyzer (IACA)*. Retrieved November 16, 2014, from http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/.

R. C. Kirby, M. Knepley, A. Logg, and L. R. Scott. 2005. Optimizing the evaluation of finite element matrices. *SIAM Journal on Scientific Computing* 27, 3, 741–758. DOI:http://dx.doi.org/10.1137/040607824

R. C. Kirby and A. Logg. 2006. A compiler for variational forms. *ACM Transactions on Mathematical Software* 32, 3, 417–444. DOI:http://dx.doi.org/10.1145/1163641.1163644

M. Knepley and A. Terrel. 2013. Finite element integration on GPUs. *ACM Transactions on Mathematical Software* 39, 2, Article No. 10. DOI:http://dx.doi.org/10.1145/2427023.2427027

F. Kruel and K. Bana. 2013. Vectorized OpenCL implementation of numerical integration for higher order finite elements. *Computers and Mathematics with Applications* 66, 10, 2030–2044. DOI:http://dx.doi.org/10.1016/j.camwa.2013.08.026

A. Logg, K.-A. Mardal, and G. N. Wells (Eds.). 2012. *Automated Solution of Differential Equations by the Finite Element Method*. Lecture Notes in Computational Science and Engineering, Vol. 84. Springer. DOI:http://dx.doi.org/10.1007/978-3-642-23099-8

F. Luporini. 2014a. Helmholtz, Advection-Diffusion, and Burgers UFL Code. Retrieved November 16, 2014, from https://github.com/firedrakeproject/firedrake/tree/pyop2-ir-perf-eval/t ests/perf-eval.

F. Luporini. 2014b. Static Linear Elasticity Code. Retrieved November 16, 2014, from https://github.com/firedrakeproject/firedrake-bench/tree/experiments/elasticity.

G. R. Markall, D. A. Ham, and P. H. J. Kelly. 2010. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science* 1, 1, 1815–1823. DOI:http://dx.doi.org/10.1016/j.procs.2010.04.203 ICCS 2010.

G. R. Markall, F. Rathgeber, L. Mitchell, N. Loriant, C. Bertolli, D. A. Ham, and P. H. J. Kelly. 2013. Performance portable finite element assembly using PyOP2 and FEniCS. In *Supercomputing*. Lecture Notes in Computer Science, Vol. 7905. Springer, 279–289.

K. B. Ølgaard and G. N. Wells. 2010. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software* 37, 1, Article No. 8. DOI:http://dx.doi.org/10.1145/1644001.1644009

M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code generation for DSP transforms. In *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation* 93, 2, 232–275.

F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. J. Kelly. 2012. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking, Storage, and Analysis*. 1116–1123. DOI:http://dx.doi.org/10.1109/SC.Companion.2012.134

D. Rossinelli, B. Hejazialhosseini, P. Hadjidoukas, C. Bekas, A. Curioni, A. Bertsch, S. Futral, S. J. Schmidt, N. A. Adams, and P. Koumoutsakos. 2013. 11 PFLOP/s simulations of cloud cavitation collapse. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'13)*. ACM, New York, NY, Article No. 3. DOI:http://dx.doi.org/10.1145/2503210.2504565

F. P. Russell and P. H. J. Kelly. 2013. Optimized code generation for finite element local assembly using symbolic manipulation. *ACM Transactions on Mathematical Software* 39, 4, Article No. 26.

J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, and P. D. Hovland. 2010. Speeding up Nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*. ACM, New York, NY, 253–262. DOI:http://dx.doi.org/10.1145/1810085.1810120

D. G. Spampinato and M. Püschel. 2014. A basic linear algebra compiler. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. 23.

K. Stock, T. Henretty, I. Murugandi, P. Sadayappan, and R. Harrison. 2011. Model-driven SIMD code generation for a multi-resolution tensor kernel. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE, Los Alamitos, CA, 1058–1067. DOI:http://dx.doi.org/10.1109/IPDPS.2011.101

Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. 2011. The Pochoir stencil compiler. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. ACM, New York, NY, 117–128. DOI:http://dx.doi.org/10.1145/1989493.1989508

P. E. J. Vos, S. J. Sherwin, and R. M. Kirby. 2010. From H to P efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *Journal of Computational Physics* 229, 13, 5161–5181. DOI:http://dx.doi.org/10.1016/j.jcp.2010.03.031

R. C. Whaley and J. J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (Supercomputing'98)*. IEEE, Los Alamitos, CA, 1–27. http://dl.acm.org/citation.cfm?id=509058.509096.

Received July 2014; revised October 2014; accepted October 2014