COFFEE: an Optimizing Compiler for Finite Element Local Assembly

GANG ZHOU, College of William and Mary
YAFENG WU, University of Virginia
TING YAN, Eaton Innovation Center
TIAN HE, University of Minnesota
CHENGDU HUANG, Google
JOHN A. STANKOVIC, University of Virginia
TAREK F. ABDELZAHER, University of Illinois at Urbana-Champaign

The numerical solution of partial differential equations using the finite element method is one of the key applications of high performance computing. Local assembly is its characteristic operation. This entails the execution of a problem-specific kernel for each element in the discretized problem domain. Since the domain size can be huge, executing efficient kernels is fundamental. Their optimization is, however, a challenging issue. Even though affine loop nests are generally present, the short trip counts and the complexity of mathematical expressions make it hard to determine a single or unique sequence of successful transformations. We present, therefore, the design and systematic evaluation of COFFEE, a domain-specific compiler for local assembly kernels. COFFEE manipulates abstract syntax trees by introducing composable optimizations aimed at improving instruction-level parallelism, especially SIMD vectorization, and register locality. It then generates C code including vector intrinsics. Experiments using a range of finite-element forms of increasing complexity show that significant performance improvement is achieved.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms: Design, Algorithms, Performance

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

ACM Reference Format:
Gang Zhou, Yafeng Wu, Ting Yan, Tian He, Chengdu Huang, John A. Stankovic, and Tarek F. Abdelzaher, 2010. A multifrequency MAC specially designed for wireless sensor network applications. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 19 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

In many fields, such as computational fluid dynamics, computational electromagnetics, and structural mechanics, phenomena are modelled by partial differential equations

This research is partly funded by the MAPDES project, by the Department of Computing at Imperial College London, by EPSRC through grants EP/I00677X/1, EP/I006761/1, and EP/L000407/1, and by NERC grants NE/K008951/1 and NE/K006789/1. The authors would like to thank Dr. Carlo Bertolli, Dr. Lawrence Mitchell, and Dr. Francis Russell for their invaluable suggestions and their contribution to the Firedrake project.
Author's addresses: G. Zhou, Computer Science Department, College of William and Mary; Y. Wu and J. A. Stankovic, Computer Science Department, University of Virginia; T. Yan, Eaton Innovation Center; T. He, Computer Science Department, University of Minnesota; C. Huang, Google; T. F. Abdelzaher, (Current address) NASA Ames Research Center, Moffett Field, California 94035.

DOI: http://dx.doi.org/10.1145/0000000.0000000

(PDEs). Numerical techniques, such as the finite volume method and finite element method, are widely employed to approximate solutions of these PDEs. Unstructured meshes are often used to discretize the equation domain, since their geometric flexibility allows solvers to be extremely effective. The solution is sought in each cell, or "element", of the discretized domain by applying suitable numerical operations. As the number of elements can be of the order of trillions [Rossinelli et al. 2013], a major issue is the time required to execute the computation, which can be hours or days. To address this problem, domain-specific languages (DSLs) have been developed. The successful port of Hydra, an industrial computational fluid dynamics application developed by Rolls Royce for turbomachinery design (based on finite volume method, roughly 50,000 lines of code and with meshes in excess of 100M edges), to OP2 [Mudalige et al. 2013], demonstrates the effectiveness of the DSL approach in the implementation of PDE solvers [Reguly et al. 2014].

OP2 adopts a programming model in which computations are expressed through self-contained functions, referred to as "kernels". A kernel is applied to all elements in a set of mesh components, such as edges, vertices, or cells, with an implicit synchronization between the application of two consecutive kernels. On commodity multicores, a kernel is executed sequentially by a thread, while parallelism is achieved partitioning the mesh and assigning each partition to a thread. Similar programming and execution models are adopted in [Markall et al. 2013], [Logg et al. 2012], [Applied Modelling and Computation Group 2010], [DeVito et al. 2011]. Kernel optimization is one of the major concerns in unstructured mesh applications. In this paper, we tackle this problem in the context of the finite element method.

We focus on improving the performance of local assembly ("assembly", in the following), a fundamental step in the finite element method that covers an important fraction of the overall computation run-time, often in the range 30-60%. During the assembly phase, the solution of the PDE is approximated by executing a suitable kernel over all elements in the discretized domain. A kernel's working set is usually small enough to fit the L1 cache; it might need L2 cache when high-order methods are employed. However, we do not consider the latter case. 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 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 minimization of floating-point operations, register allocation and instruction-level parallelism, especially in the form of SIMD vectorization.

Achieving high-performance 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. The variation in loop trip counts and their typically small value further exacerbates the issue. A compiler-based approach is, therefore, the only reasonable option to obtain close-to-peak performance in a wide range of different local assembly kernels. Optimizations like generalized loop-invariant code motion, vector-register tiling, and expression splitting, as well as their composition, are essential, but none are supported by state-of-the-art polyhedral and vendor compilers. BLAS routines could theoretically be employed, although a fairly complicated control- and data-flow analysis is required to automate identification and extraction of matrix-matrix multiplies. In addition, as detailed in Section 5.6, the small dimension of the matrices involved and the potential loss in data locality can limit or eliminate the performance gain of this approach.

ALGORITHM 1: General structure of a local assembly kernel generated by Firedrake.

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

In order 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 [fir 2013], a system for solving PDEs through the finite element method based on the PyOP2 abstraction [Markall et al. 2013]. It supports all problems expressible with this framework. This allows us to evaluate our code transformations in a range of real-world problems, varying two key parameters that impact both 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).

Our experiments show that the original generated code for non-trivial assembly kernels is sub-optimal. Our cost-model-driven sequence of source-to-source code transformations, aimed at improving SIMD vectorization and register data locality, can result in performance improvements up to $1.5\times$ over code in which only basic transformations have been performed, such as generalized loop-invariant code motion, padding, and data alignment, and up to $4.4\times$ over original kernels. The contributions of this paper are

(1) An optimization strategy for finite element local assembly. Our approach exploits domain knowledge and goes beyond the limits of both vendor and research compilers.
(2) Design and implementation of a compiler that automates the proposed code transformations for any problems expressible in Firedrake.
(3) Systematic analysis using a suite of examples of real-world importance, and evidence of significant performance improvements on two Intel architectures, a Sandy Bridge CPU and the Xeon Phi.

The paper is organized as follows. In Section 2 we provide some background on local assembly, showing code generated by Firedrake and emphasizing 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. Related work is illustrated in Section 6, while Section 7 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 PDE solution. The process consists of numerically evaluating problem-specific integrals to produce a matrix and a vector ([Olgaard and Wells 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.

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

ALGORITHM 2: Local assembly 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]);
}
```

ALGORITHM 3: Local assembly 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]);
 }
}
```

Given a finite element description of the input problem, expressed through the domain-specific Unified Form Language [Alnæs et al. 2014] (UFL), Firedrake generates a C-code kernel implementing assembly using a numerical integration algorithm. The same kernel can be applied to any elements: this descends from the mathematical formulation of assembly, which states that the evaluation of a local matrix can be reduced to integrating 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 stated, the subject of

this paper 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 two dimensional array used to store the element matrix, the element's coordinates in the discretized domain, and coefficient fields, for instance indicating the values of velocity or pressure in the element. The output is the evaluated element matrix. 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, intuitively, to interpolate the contribution to the PDE solution over the element. The choice of the basis functions is expressed in UFL directly by users. In the generated code, they are represented as global read-only two dimensional arrays (i.e. using `static const` in C) of double precision floats; 3) evaluation of the element matrix in an affine loop nest, in which the integration is performed. Table 2 shows the variable names we will use in the upcoming code snippets to refer to the various kernel 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	double**	w
Numerical integration weights	double[]	W
Basis functions (and derivatives)	double[][]	X, Y, X1, ...
Element matrix	double[][]	A

Table I. Type and variable names used in the various code snippets to identify local assembly 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, it 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 Figure 2, which shows the assembly kernel for a Helmholtz problem using Lagrange basis functions on 2D elements with polynomial order $p = 1$. In other scenarios, for instance when solving the Burgers equation, whose assembly code is given in Figure 3, the number of arrays involved in the computation of the element matrix can be much larger: in this case, 14 unique arrays are accessed, and the same array can be referenced multiple times within the 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 can be larger (proportionally 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, along with the Diffusion problem,

Despite the infinite variety of problem-specific assembly kernels which 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 stride-1; 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; 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 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. As already emphasized, the structure of mathematical expressions evaluating the element matrix and the variation in loop trip counts, although typically limited to the order of tens of iterations, render the optimization process challenging. It is not always the same set of optimizations that bring performance closest to the machine peak. For example, the Burgers problem in Figure 3, given the large number of arrays accessed, suffers from high register pressure, whereas the Helmholtz problem in Figure 2 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 local assembly kernels, and to what extent their composability is effective in a class of problems and architectures.

3.1. Prerequisites for Effective Auto-vectorization: Padding and Data Alignment

Auto-vectorization of assembly code computing the element matrix can be less effective if data are not aligned and if the length of the innermost loop is not a multiple of the vector length vl. Data alignment is enforced in two steps. Initially, 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 256 bit long vector registers and 64-bit double-precision floating-point values). In this case, a padded version of the array will have size 3×4. The compiler is explicitly informed about data alignment using a suitable pragma. Padding of all two dimensional arrays involved in the evaluation of the element matrix also allows us to safely round the loop trip count to the nearest multiple of vl. This avoids the introduction

ALGORITHM 4: Local assembly code for the Helmholtz problem in Figure 2 after application of padding, data alignment, and *licm*, for an AVX architecture. In this example, sub-expressions 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];
 }
}
```

of a remainder (scalar) loop from the compiler, which would render vectorization less efficient.

3.2. Generalized Loop-invariant Code Motion

From the inspection of the codes in Figures 2 and 3, it can be noticed that the computation of A involves evaluating many sub-expressions which only depend on two iteration variables. Since symbols in most of these sub-expressions are read-only variables, there is ample space 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 sub-expressions that are invariant with respect to the innermost loop. This is an issue for sub-expressions depending on i-k, which are not automatically lifted in the loop order `ijk`. Second, the hoisted code is scalar, i.e. it is not subjected to auto-vectorization. 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 (e.g. in Helmholtz), invariant sub-expressions along j are identical to those along k, and both loops iterate over the same iteration space. In these cases, we safely avoid redundant pre-computation since, as anticipated in Section 2, a property of our domain is that j and k loops share the same iteration space.

Figure 4 shows the Helmholtz assembly 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 remarkably 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 Figure 4 and we ignore the presence of the operation `det*W3[i]`, the computation of the element matrix is abstractly expressible as

$$A_{jk} = \sum_{\substack{x \in B' \subseteq B \\ y \in B'' \subseteq B}} x_j \cdot y_k \qquad j,k = 0, ..., 3 \tag{1}$$

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 1 illustrates how we can evaluate 16 entries ($j, k = 0, ..., 4$) 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 (*gnu* and *intel*), 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 in 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 2, executed once outside of the `ijk` loop nest. The generated pseudo-code for the simple Helmholtz problem when using *op-vect* is shown in Figure 5.

3.4. Expression Splitting

In complex kernels, like Burgers in Figure 3, 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 to be obtained. 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

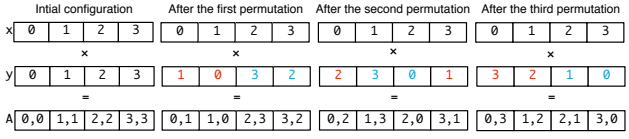

Fig. 1. Outer-product vectorization by permuting values in a vector register.

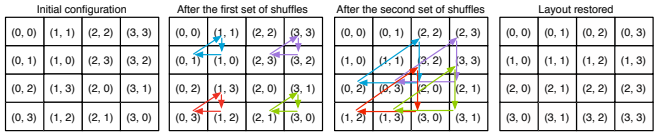

Fig. 2. Restoring the storage layout after *op-vect*. The figure shows how 4×4 elements in the top-left block of the element matrix A can be moved to their correct positions. Each rotation, represented by a group of three same-colored arrows, is implemented by a single shuffle intrinsic.

ALGORITHM 5: Local assembly code generated by Firedrake for the Helmholtz problem after application of *op-vect* on top of the optimizations shown in Figure 4. Here, we assume the polynomial order is $p = 2$, since *op-vect* can not be used when an iteration space dimension is smaller than the vector length. The original size of the j-k iteration space (i.e. before padding was applied) was 6×6. In this example, the unroll-and-jam factor is 1.

```
void helmholtz(double A[8][8], double **coords) {
 // K, det = Compute Jacobian (coords)
 // Declaration of basis function matrices

 for (int i = 0; i<6; i++) {
  // Do loop-invariant code motion
  for (int j = 0; j<4; j+=4)
   for (int k = 0; k<8; k+=4) {
    // load and set intrinsics
    // Compute A[1,1],A[2,2],A[3,3],A[4,4]
    // One permute_pd intrinsic per k-loop load
    // Compute A[1,2],A[2,1],A[3,4],A[4,3]
    // 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<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 A[j+1][k], A[j+2][k], A[j+3][k]
  }
 }
}
```

motion to Burgers on a 3D mesh requires 33 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 in Figure 6.

ALGORITHM 6: Local assembly code generated by Firedrake for the Helmholtz problem in which *split* has been applied on top of the optimizations shown in Figure 4. In this example, the split factor is 2.

```
void helmholtz(double A[3][4], double **coords) {
  // Same code as in Figure 4 up to line 15
  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*W3[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*W3[i];
}
```

Splitting an expression has three drawbacks. Firstly, it increases the number of accesses to A proportionally to the "split factor", which is the number of sub-expressions produced. Secondly, depending on how the split is executed, it can lead to redundant computation; for example, the times the product $det * W3[i]$ is performed is proportional to the number of sub-expressions, as shown in the code snippet. Finally, it might affect register locality: for instance, the same array could be accessed in different sub-expressions, requiring a proportional number of loads be performed. This is not the case of the Helmholtz example. Nevertheless, as shown in Section 5, the performance gain from improved register reuse along inner dimensions can still be greater, especially if the split factor and the splitting itself use heuristics to minimize the aforementioned issues.

Table II summarizes the code transformations described so far. Given that many of these transformations depend on some parameters (e.g. tile size), we need a mechanism to prune such a large space of optimization. This aspect is treated in Section 4.

Name (Abbreviation)	Parameter
Generalized loop-invariant code motion (*licm*)	
Padding	
Data Alignment	
Loop interchange	loops
Loop unrolling	unroll factor
Outer-product vectorization (*op-vect*)	tile size
Expression splitting (*split*)	split point, split factor

Table II. Overview of code transformations for Firedrake-generated assembly kernels.

ALGORITHM 7: Pseudocode of the COFFEE pipeline.

```
The COFFEE Compiler
Input: ast, wrapper, isa
Output: code
// Analyze ast and build optimization plan
it_space = analyze(ast)
if not it_space then
  | ast.apply_inter_kernel_vectorization(wrapper, isa)
  | return wrapper + ast.from_ast_to_c()
end
plan = cost_model(it_space.n_inner_arrays, isa.n_regs)
// Optimize ast based on plan
ast.licm()
ast.padding()
ast.data_align()
if plan.permute then
  | ast.permute_assembly_loops()
end
if plan.sz_split then
  | ast.split(plan.sz_split)
end
if plan.uaj_factor then
  | uaj = MIN(plan.uaj_factor, ⌊it_space.j.size/isa.vf⌋)
  | ast.op_vect(uaj)
end
return wrapper + ast.from_ast_to_c()
```

4. OVERVIEW OF COFFEE

Firedrake users employ the Unified Form Language to write problems in a notation resembling mathematical equations. At run-time, the high-level specification is translated by a modified version of the FEniCS Form Compiler [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, which is capable of applying the transformations described in Section 3. 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. Because of the large number of parametric transformations, COFFEE needs a mechanism to select the most suitable optimization strategy for a given problem. Auto-tuning might be used, although it would significantly increase the run-time overhead, since the generation of ASTs occurs at run-time as soon as problem-specific data are available. Our optimization strategy, based on heuristics and a simple cost model, is described in the following, along with an overview of the compiler.

The compiler structure is outlined in Figure 7. Initially the AST is inspected, looking for the presence of iteration spaces and domain-specific information provided by the higher layer. If the kernel lacks an iteration space, then so-called inter-kernel vectorization, in which the outer, non-affine loop over mesh elements is vectorized, can be applied. This feature, currently under development, has been shown to be useful in several finite volume applications [Reguly et al. 2007]. Subsequently, an ordered sequence of optimization steps are executed. Application of *licm* must precede padding and data alignment, due to the introduction of temporary arrays. Based on a cost model, loop interchange, *split*, and *op-vect* may be introduced. Their implementation is based on analysis and transformation of the kernel's 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 auto-vectorization, is inserted.

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 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 element matrix.

Since loop unroll and unroll-and-jam of outer loops are fundamental to the exposure of instruction-level parallelism, tuning critical parameters such as the unroll factor is of great importance. However, inspecting assembly code and comparing with other hand-written implementations indicates that recent versions of the Intel compiler are capable of estimating close-to-optimal values for such parameters. This is particularly true for assembly kernels, in which the loop nest is affine, bounds are usually very small and known at compile-time, and memory accesses are stride one. We therefore leave the backend compiler in charge of selecting the unroll factor. This choice also simplifies COFFEE's cost model. The only situation in which we explicitly unroll-and-jam a loop is when *op-vect* is used, since the transformed code seems to prevent the Intel compiler from applying this optimization, even if specific pragmas are added.

The cost model is shown in Figure 8. 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 3-8), which is approximated as half of the total (line 2). A split factor of n means that the assembly expression should be "cut" into n sub-expressions. Depending on the structure of the assembly expression, each sub-expression might end up accessing a different number of arrays; the cost model is simplified by assuming that all sub-expressions 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, i.e. those not used for storing hoisted terms (line 9-11). Finally, the profitability of loop interchange is evaluated (line 12-16).

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, which can impact the effectiveness of the optimization strategy. The `icc 13.1` compiler was used. On the Sandy Bridge, the compilation flags used were `-O2` and `-xAVX` for auto-vectorization. On the Xeon Phi, optimization level `-O3` was used. Other optimization levels performed, in general, slightly worse. Our code transformations were evaluated in three real-world problems based on the following PDEs:

ALGORITHM 8: 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 auto-tuning.

```
Cost Model
Input: n_outer_arrays, n_inner_arrays, n_consts, n_regs
Output: uaj_factor, split_factor
n_outer_regs = n_regs / 2
split_factor = 0
// Compute spltting factor
while n_outer_arrays > n_outer_regs do
  | n_outer_arrays = n_outer_arrays / 2
  | split_factor = split_factor + 1
end
// Compute unroll-and-jam factor for op-vect
n_regs_avail = n_regs - (n_outer_arrays + n_consts)
uaj_factor = ⌈n_reg_avail / n_inner_arrays⌉
if n_outer_arrays > n_inner_arrays then
  | permute = True
else
  | permute = False
end
return <permute, split_factor, uaj_factor>
```

(1) Helmholtz
(2) Advection-Diffusion
(3) Burgers

The code was written in UFL (available at [ufl 2014]) and then executed over real unstructured meshes through Firedrake. The Helmholtz code has already been shown in Figure 2. For Advection-Diffusion, the Diffusion equation, which uses the same differential operators as Helmholtz, is considered. In the Diffusion kernel, the main differences with respect to Helmholtz are the absence of the Y array and the presence of a few more constants for computing the element matrix A. Burgers is a non-linear problem employing differential operators different from those of Helmholtz, which has a major impact on the generated assembly code (see Figure 3), where a larger number of basis function matrices ($X1$, $X2$, ...) and constants ($F0$, $F1$, ..., $K0$, $K1$, ...) are needed.

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 larger the element shape and p, the larger the iteration space. Triangles, tetrahedron, 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, respectively, 3, 4, and 6. Moving to bigger shapes has the effect of increasing the number of basis function arrays, since, intuitively, the behavior of the equation has now to be approximated also along a third axis. On the other hand, the polynomial order affects only the problem size (the three loops i, j, and k, and, as a consequence, the size of X and Y arrays). A range of polynomial orders from $p = 1$ to $p = 4$ 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 $p = 4$.

In the following, results for the loop order `ijk` are shown. For the considerations exposed in Section 4, loop interchanges having an inner loop over i caused slow downs; also, interchanging j and k loops while keeping i as outermost loop did not provide any benefits.

5.2. Impact of Generalized Loop-invariant Code Motion

Table III illustrates the performance improvement obtained on the Sandy Bridge and Xeon Phi machines when *licm*, data alignment, and padding are used. We distinguish between *licm* and *licm-ap*; the latter makes use of padding and data alignment. Inspection of assembly code generated by `icc` confirmed all limitations described in Section 3.2: only sub-expressions invariant to the inner loop are hoisted and their execution is not vectorized. Padding and data alignment enhance, in general, the quality of auto-vectorization. Sometimes the run-time of *licm-ap* is similar to that of *licm* because the element matrix size is already a multiple of the vector length, so no scalar reminder loop has to be introduced. Occasionally *licm-ap* is slower than *licm* (e.g. in Burgers $p = 3$ on the Sandy Bridge). One possible explanation is that the large number of aligned temporaries introduced by *licm* induce cache associativity conflicts.

5.3. Impact of Vector-register Tiling

Figures 3 and 5 show the performance achieved applying *op-vect* on top of *licm-ap* to the Helmoltz and Diffusion kernels on the Sandy Bridge, whereas Figures 4 and 6 illustrate analogous results for the Xeon Phi. For each problem instance we report two bars: one indicates the best run-time obtained by auto-tuning the unroll/unroll-and-jam factors, whereas the other shows the result retrieved with COFFEE's cost model. In general, there is no substantial difference between the two. This is chiefly because these kernels fit the L1 cache, so, within a certain degree of confidence, it is possible to predict the fastest implementation based on register allocation.

The rationale behind these results is that, for smaller configurations, vector-register tiling is marginally helpful, while its impact is the dominant factor for larger problems. This is because in the considered kernels the number of accessed arrays along the innermost loop dimension is rather small (between 2 and 4), so extensive unrolling is often enough to maximize register re-use when the loops are relatively short. On the other hand, as the iteration space becomes bigger, vector-register tiling leads to improvements up to 1.4× on the Sandy Bridge (Diffusion, prismatic mesh, $p = 4$) and up to 1.4× on the Xeon Phi (Helmholtz, tetrahedral mesh, $p = 3$). Using the Intel Architecture Code Analyzer tool [IAC] on the Sandy Bridge, we confirmed that this is a consequence of increased register re-use. 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 30.4 GFlop/s. In the case of Diffusion using a prismatic mesh and $p = 4$, 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.

5.4. Impact of Expression Splitting

Expression splitting relieves the register pressure when the element matrix evaluation reads from a large number of arrays, at the price of increasing accesses to the

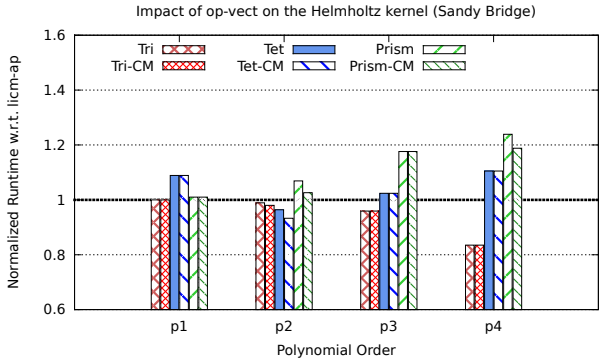

Fig. 3. Performance improvement obtained applying *op-vect* on top of *licm-ap* to the Helmholtz kernel on the Sandy Bridge. CM refers to where the cost model has been used to determine the sequence of code transformations.

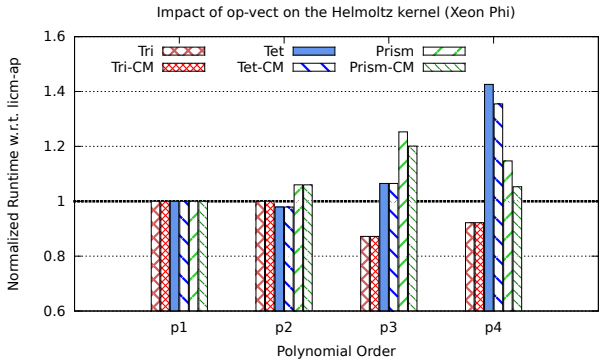

Fig. 4. Performance improvement obtained applying *op-vect* on top of *licm-ap* to the Helmholtz kernel on the Xeon Phi. CM refers to where the cost model has been used to determine the sequence of code transformations.

element matrix itself and potentially affecting data locality, as detailed in Section 3.4. It is not particularly useful, therefore, for Helmholtz and Diffusion, where only between 4 and 8 basis functions or temporaries need to be read at every iteration. Slow downs up to 1.4× and up to 1.6× were observed, respectively, on the Sandy Bridge and the Xeon Phi. The cost model, however, prevents the adoption of the transformation: the `while` statement at line 5 in Figure 8 is indeed never entered. For the Burgers kernel, *split* has a key role on the Sandy Bridge, whereas it has only a minimal impact on the Xeon Phi. Figure 7 and 8 show the improvement due to using *split* over the fastest implementation between *licm* and *licm-ap*. On the Sandy Bridge, in almost all test 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. On the contrary, the larger number of vector units present on the Xeon Phi relieve register spilling, making *split* only marginally useful for loop unrolling.

On the Sandy Bridge, the performance of the Burgers kernel on a prismatic mesh was 20.0 GFlop/s from $p = 1$ to $p = 3$, while it was 21.3 GFlop/s in the case of $p = 4$. These values are notably close to the peak performance of 30.4 GFlop/s. Disabling *split*

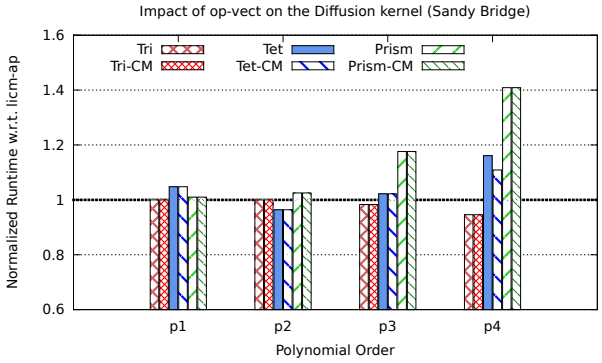

Fig. 5. Performance improvement obtained applying *op-vect* on top of *licm-ap* to the Diffusion kernel on the Sandy Bridge. CM refers to where the cost model has been used to determine the sequence of code transformations.

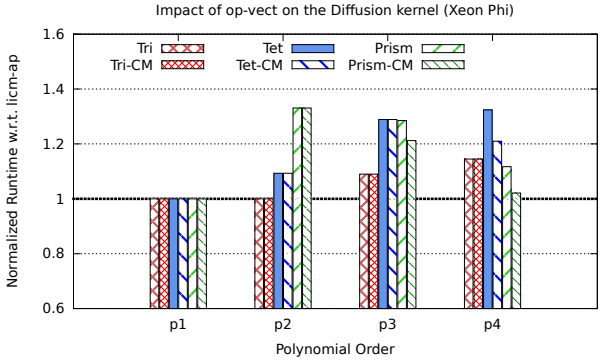

Fig. 6. Performance improvement obtained applying *op-vect* on top of *licm-ap* to the Diffusion kernel on the Xeon Phi. CM refers to where the cost model has been used to determine the sequence of code transformations.

makes the performance drop to 17.0 GFlop/s for $p = 1, 2$, 18.2 GFlop/s for $p = 3$, and 14.3 GFlop/s for $p = 4$. These values are in line with the speedups shown in Figure 7.

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 for two reasons: for small split factors, such as 1 and 2, the data space to be tiled is still too big, and register spilling affects run-time; for higher ones, sub-expressions become so small that, as explained in Section 5.3, full or partial unrolling of the loop nest achieves a certain degree of register re-use by itself.

5.5. Comparison with FEniCS Form Compiler's built-in Optimizations

We have modified the FEniCS Form Compiler (FFC) to return an abstract syntax tree representation of a local assembly kernel, rather than plain C++ code, so as to enable code transformations in COFFEE. Besides Firedrake, FFC is used in the FEniCS project [Logg et al. 2012]. In FEniCS, FFC can apply its own model-driven optimizations to local assembly kernels [Olgaard and Wells 2010], which mainly consist of loop-invariant code motion and elimination, at code generation time, of floating point

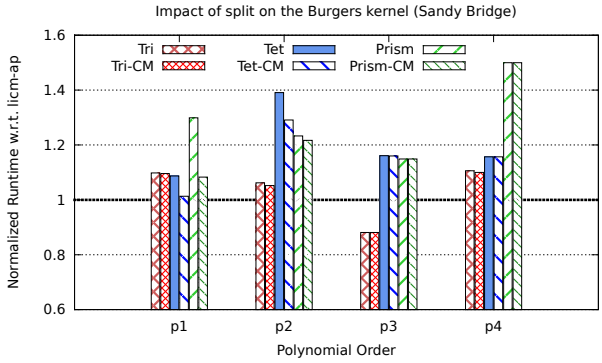

Fig. 7. Performance improvement obtained applying *op-vect* on top of *licm-ap* to the Burgers kernel on the Sandy Bridge. CM refers to the cases in which the cost model has been used to determine the sequence of code transformations.

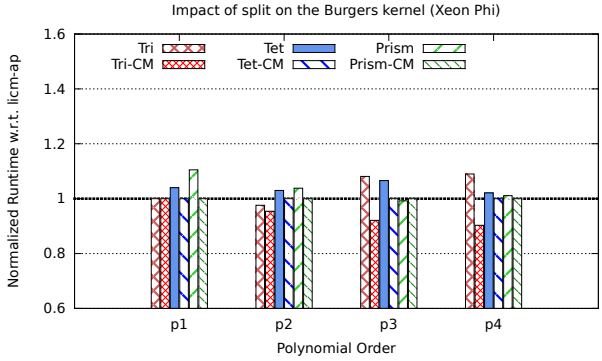

Fig. 8. Performance improvement obtained applying *op-vect* on top of *licm-ap* to the Burgers kernel on the Xeon Phi. CM refers to the cases in which the cost model has been used to determine the sequence of code transformations.

operations involving zero-valued entries in basis function arrays. The FEniCS Form Compiler's loop-invariant code motion is, however, different from COFFEE's, since it is based on expansion of arithmetic operations, for example applying distributivity and associativity to products and sums at code generation time, to identify terms that are invariant of the whole loop nest. In addition, depending on the way expansion is performed, operation count often does not decrease significantly. On the other hand, elimination of zero-valued terms, which are the result of certain properties of 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 FEniCS' by avoiding non-contiguous 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 best FEniCS (FFC) implementation (i.e. the combination of optimizations for each problem which results in the fastest code) on the Sandy Bridge for the Burgers, Helmholtz and Diffusion kernels. Burgers' slow downs 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 figure shown is for the worst slow down, which was obtained with a triangular mesh and $p = 1$.

This is a result of removing zero-valued entries in FEniCS' basis functions: some operations are avoided, but indirection arrays prevent auto-vectorization, 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 are, respectively, $1.10\times$ and $1.18\times$ (2D mesh, $p = 1$), which tend to increase with polynomial order and element shape up to the values illustrated in the table.

5.6. Comparison with hand-made 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 slower than the case in which *licm-ap* was used, with the slow down being even worse for smaller problem sizes. These experiments suggest that the question regarding to what extent linear algebra libraries or, analogously, auto-tuning strategies 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 external tools 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.

6. 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 [Applied Modelling and Computation Group 2010], and of course Firedrake. Numerical integration is usually employed to implement the local assembly phase. The recent introduction of 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] have been implemented. We have designed COFFEE to specifically target numerical integration because it has been demonstrated that it remains the optimal choice for a wide class of problems [Olgaard and Wells 2010].

Optimization of local assembly using numerical integration for CPU platforms has been addressed in FEniCS [Olgaard and Wells 2010]. The comparison between COFFEE and this work is presented in Section 5.5. In [Markall et al. 2010], and more recently in [Knepley and Terrel 2013], the 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.

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 et al. 2005] for fast fourier transforms. In both cases, pruning the implementation space is fundamental to mitigate complexity and overhead. Likewise, COFFEE uses a cost model and heuristics (Section 4) to steer the optimization process.

7. CONCLUSIONS

In this paper, we have presented design, optimizations and systematic performance evaluation of COFFEE, a compiler for finite element local assembly. In this context, to the best of our knowledge, COFFEE is the first compiler capable of introducing low-level optimizations to maximize instruction-level parallelism, register locality and SIMD vectorization. Assembly kernels have peculiar 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 two state-of-the-art Intel platforms. Further work include a comprehensive study about feasibility and constraints on transforming kernels into a sequence of calls to external linear algebra libraries. COFFEE supports all of the problems expressible in Firedrake, and it is already integrated with this framework.

REFERENCES

Intel architecture code analyzer. [Online]. Available: http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/.

2013. The Firedrake Project. http://www.firedrakeproject.org. (2013).

2014. Helmholtz, Advection-Diffusion, and Burgers UFL code. https://github.com/firedrakeproject/firedrake/tree/pyop2-ir-perf-eval/tests/perf-eval. (2014).

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 Trans Math Software* 40, 2, Article 9 (2014), 9:1–9:37 pages. DOI:http://dx.doi.org/10.1145/2566630

Applied Modelling and Computation Group 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.

Uday Bondhugula, Albert 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, USA, 101–113. DOI:http://dx.doi.org/10.1145/1375581.1375595

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. 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)*. ACM, New York, NY, USA, Article 9, 12 pages. DOI:http://dx.doi.org/10.1145/2063384.2063396

Matteo Frigo, Steven, and G. Johnson. 2005. The design and implementation of FFTW3. In *Proceedings of the IEEE*. 216–231.

Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A Stencil Compiler for Short-vector SIMD Architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 13–24. DOI:http://dx.doi.org/10.1145/2464996.2467268

Robert C. Kirby and Anders Logg. 2006. A Compiler for Variational Forms. *ACM Trans. Math. Softw.* 32, 3 (Sept. 2006), 417–444. DOI:http://dx.doi.org/10.1145/1163641.1163644

Matthew G. Knepley and Andy R. Terrel. 2013. Finite Element Integration on GPUs. *ACM Trans. Math. Softw.* 39, 2, Article 10 (Feb. 2013), 13 pages. DOI:http://dx.doi.org/10.1145/2427023.2427027

Filip Kruel and Krzysztof Bana. 2013. Vectorized OpenCL Implementation of Numerical Integration for Higher Order Finite Elements. *Comput. Math. Appl.* 66, 10 (Dec. 2013), 2030–2044. DOI:http://dx.doi.org/10.1016/j.camwa.2013.08.026

Anders Logg, Kent-Andre Mardal, Garth N. Wells, and others. 2012. *Automated Solution of Differential Equations by the Finite Element Method*. Springer. DOI:http://dx.doi.org/10.1007/978-3-642-23099-8

Graham R. Markall, David A. Ham, and Paul H.J. Kelly. 2010. Towards generating optimised finite element solvers for {GPUs} from high-level specifications. *Procedia Computer Science* 1, 1 (2010), 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 *Proceedings of the International Supercomputing Conference (ISC) '13 (Lecture Notes in Computer Science)*, Vol. 7905. In press.

G.R. Mudalige, M.B. Giles, J. Thiyagalingam, I.Z. Reguly, C. Bertolli, P.H.J. Kelly, and A.E. Trefethen. 2013. Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems. *Parallel Comput.* 39, 11 (2013), 669 – 692. DOI:http://dx.doi.org/10.1016/j.parco.2013.09.004

Kristian B. Olgaard and Garth N. Wells. 2010. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Softw.* 37, 1, Article 8 (Jan. 2010), 23 pages. DOI:http://dx.doi.org/10.1145/1644001.1644009

James W. Lottes Paul F. Fischer and Stefan G. Kerkemeier. 2008. 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. 2005. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93, 2 (2005), 232– 275.

I. Z. Reguly, E. László, G. R. Mudalige, and M. B. Giles. 2007. Vectorizing Unstructured Mesh Computations for Many-core Architectures. In *Proceedings of Programming Models and Applications on Multicores and Manycores (PMAM'14)*. ACM, New York, NY, USA, Article 39, 12 pages. DOI:http://dx.doi.org/10.1145/2560683.2560686

Istvan Z. Reguly, Gihan R. Mudalige, Carlo Bertolli, Michael B. Giles, Adam Betts, Paul H. J. Kelly, and David Radford. 2014. Acceleration of a Full-scale Industrial CFD Application with OP2. (March 2014). http://arxiv.org/abs/1403.7209v1.

Diego Rossinelli, Babak Hejazialhosseini, Panagiotis Hadjidoukas, Costas Bekas, Alessandro Curioni, Adam Bertsch, Scott Futral, Steffen J. Schmidt, Nikolaus A. Adams, and Petros 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, USA, Article 3, 13 pages. DOI:http://dx.doi.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 Trans. Math. Software* 39, 4 (????).

Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul 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, USA, 253–262. DOI:http://dx.doi.org/10.1145/1810085.1810120

Daniele G. Spampinato and Markus Püschel. 2014. A Basic Linear Algebra Compiler. In *International Symposium on Code Generation and Optimization (CGO)*.

Kevin Stock, Tom Henretty, Iyyappa Murugandi, P. Sadayappan, and Robert Harrison. 2011. Model-Driven SIMD Code Generation for a Multi-resolution Tensor Kernel. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS '11)*. IEEE Computer Society, Washington, DC, USA, 1058–1067. DOI:http://dx.doi.org/10.1109/IPDPS.2011.101

Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir Stencil Compiler. In *Proceedings of the Twenty-third Annual ACM Symposium*

on Parallelism in Algorithms and Architectures (SPAA '11). ACM, New York, NY, USA, 117–128. DOI:http://dx.doi.org/10.1145/1989493.1989508

R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (Supercomputing '98)*. IEEE Computer Society, Washington, DC, USA, 1–27. http://dl.acm.org/citation.cfm?id=509058.509096

		Sandy Bridge								Xeon Phi					
		licm				licm-ap				licm				licm-	
		p1	p2	p3	p4	p1	p2	p3	p4	p1	p2	p3	p4	p1	p2
Helmholtz	triangle	1.05	1.46	1.68	1.67	1.32	1.88	2.87	4.13	1,49	1,06	1,05	1,17	1,50	2,41
Helmholtz	tetrahedron	1.36	2.10	2.64	2.27	1.35	3.32	2.66	3.27	1,28	1,29	2,05	1,73	1,41	1,50
Helmholtz	prism	2.16	2.28	2.45	2.06	2.63	2.74	2.43	2.75	1,04	2,26	1,93	1,64	2,38	2,47
Diffusion	triangle	1.09	1.68	1.97	1.64	1.38	1.99	3.07	4.28	1,07	1,06	1,18	1,16	1,08	1,88
Diffusion	tetrahedron	1.30	2.20	3.12	2.60	1.41	3.70	3.18	3.82	1,00	1,38	2,02	1,74	1,05	1,51
Diffusion	prism	2.15	1.82	2.71	2.32	2.55	3.13	2.73	2.69	1,11	2,16	1,85	2,83	2,41	2,52
Burgers	triangle	1.53	1.81	2.68	2.46	1.56	2.28	2.61	2.77	1,21	1,42	2,34	2,97	2,84	2,26
Burgers	tetrahedron	1.61	2.24	1.69	1.59	1.61	2.10	1.60	1.78	1,01	2,55	0,98	1,21	1,48	3,83
Burgers	prism	2.11	2.20	1.66	1.32	2.19	2.32	1.64	1.42	1,39	1,56	1,18	1,04	2,18	2,82

Table III. Performance improvement due to generalized loop-invariant code motion (*licm* columns) on the Helmholtz, Diffusion and Burgers problems, varying the element shape (triangle, tetrahedron, prism) and the polynomial order ($p \in [1, 4]$), using the Lagrange element family, over

Problem	Max slow down	Max speed up
Helmholtz	-	4.14×
Diffusion	-	4.28×
Burgers	2.24×	1.61×

Table IV. Performance comparison between FEniCS (optimizations enabled) and COFFEE on the Sandy Bridge.

Online Appendix to: COFFEE: an Optimizing Compiler for Finite Element Local Assembly

GANG ZHOU, College of William and Mary
YAFENG WU, University of Virginia
TING YAN, Eaton Innovation Center
TIAN HE, University of Minnesota
CHENGDU HUANG, Google
JOHN A. STANKOVIC, University of Virginia
TAREK F. ABDELZAHER, University of Illinois at Urbana-Champaign

DOI: http://dx.doi.org/10.1145/0000000.0000000