COFFEE: an Optimizing Compiler for Finite Element Local Assembly

Fabio Luporini*, Ana Lucia Varbanescu†, Florian Rathgeber*, Gheorghe-Teodor Bercea*,
J. Ramanujam‡, David A. Ham§, and Paul H. J. Kelly*
*Imperial College London, Department of Computing,
§Imperial College London, Department of Computing and Department of Mathematics,
†TU Delft, Faculty of Engineering, Mathematics and Computer Science (EWI)
*{f.luporini12|f.rathgeber10|gheorghe-teodor.bercea08|p.kelly|david.ham}@imperial.ac.uk,
†a.l.varbanescu@uva.nl, ‡jxr@ece.lsu.edu

Abstract—**Numerical solution of partial differential equations using the finite element method is one of the key applications of high performance computing. The local assembly is its characteristic operation, which executes 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 suite of real-world examples show that significant performance improvement is achieved.**

Keywords—Finite element integration, local assembly, compilers, optimizations, simd vectorization

I. Introduction

In many fields, such as computational fluid dynamics, computational electromagnetics, and structural mechanics, phenomena are modelled by means of partial differential equations (PDEs). Numerical techniques, like finite volume method and finite element method, are widely-employed to approximate solutions of 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 element of the discretized domain by applying suitable numerical operations. As the number of elements can be of the order of millions, 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 porting of Hydra, a computational fluid dynamics industrial application devised by Rolls Royce for turbomachinery design (based on Finite Volume Method, roughly 50000 lines of code and mesh sizes that can be over 100 millions edges), to OP2 [17], demonstrates the effectiveness of the DSL approach for implementing PDEs solvers [22].

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 multi-cores, 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 [14], [4], [6]. Kernel optimization is one of the major concerns in unstructured mesh applications. In this paper, we tackle this problem by proposing an optimization strategy for a class of kernels used in finite element methods.

We focus on local assembly ("assembly", in the following), a fundamental step of a 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 to improve the accuracy of the solution. However, we do not consider the latter case. An assembly kernel is characterized by the presence of an affine, often non-perfect loop nest, where individual loops are rather small: their trip count rarely exceeds 30, and may be as low as 3 for lower 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, which prevents the adoption of standard techniques for cache locality, 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 they are not supported by state-of-the-art polyhedral and vendor compilers. BLAS routines could be theoretically employed, although a fairly complicated control- and data-flow analysis would be required to automate identification and extraction of matrix-matrix multiplies. In addition, as detailed in Section V, in which manual BLAS-based implementations are evaluated, the small dimension of the involved matrices and loss in data locality can limit the performance gain, if any.

Due to the constraints of available compilers and linear-algebra specialized 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 [2], a system for solving PDEs through the finite element method based on the PyOP2 abstraction [15]. It supports any problems expressible with this framework. This allows us to evaluate our code transformations in a range of real-world problems, varying key parameters that impact both solution accuracy and kernel cost, namely the polynomial order of the method (from $p = 1$ to $p = 4$) and the geometry of elements in the discretized domain (2D triangle, 3D tetrahedron, 3D prism).

Early experiments showed that Firedrake-generated code for non-trivial assembly kernels was 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× over "softly-optimized" code (i.e. code where only basic transformations are performed, such as generalized loop-invariant code motion, padding, and data alignment), and up to 4.44× over original kernels. The contributions of this paper are threefold

- An optimisation strategy for finite element local assembly. Our approach exploits domain knowledge and goes beyond the limits of both vendor and research compilers.
- Design and implementation of a compiler that automates the proposed code transformations for any problems expressible in Firedrake.
- Systematic evaluation 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 II we provide some background on local assembly, showing code generated by Firedrake and emphasizing critical computational aspects. Section III 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 IV. Section V shows performance results. Related work are illustrated in Section VI, while Section VII concludes the paper.

II. Background

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 an integral to produce a small dense element matrix or an element vector [18], [4]. 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.

```
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
  - Declare of constant data: basis functions and derivatives
  - Compute element matrix using numerical quadrature
```

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

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

  static const double W3[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*W3[i]);
}
```

Fig. 2. Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh with Lagrange $p = 1$ elements.

Given a mathematical description of the input problem, expressed through the domain-specific Unified Form Language [3], Firedrake generates C-code kernels implementing assembly using a numerical integration algorithm. It then triggers compilation of such kernels using an available vendor compiler, and eventually manages parallel execution over the mesh. The subject of this paper is to enhance this execution model by adding an optimization stage prior to the generation of C code. The code transformations described next are also generalizable to non-Firedrake assembly kernels, provided that numerical integration is used.

The general structure of a Firedrake-generated kernel is shown in Figure 1. Input are a pointer to a zero-initialized two dimensional array used to store the element matrix, a pointer to the element's coordinates in the discretized domain, and pointers to data of coefficient fields, for instance indicating value of velocity or pressure in the element. The output is the evaluated element matrix. A local assembly kernel can be logically split into three parts: 1) calculation of the Jacobian matrix, its determinant and its inverse; 2) definition of some constant two dimensional arrays, called "basis functions" (and their derivatives) in finite element terminology, initialized to problem-specific double-precision floating point values; 3) evaluation of the element matrix in an affine loop nest. All elements in the discretized domain share the same basis function arrays, so they are declared as global read-only arrays (i.e. using `static const` in C language). Table I shows the variable names we will use in the upcoming code snippets to refer to the various kernel's objects.

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

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

  static const double W5[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*W5[i]);
  }
}
```

Fig. 3. Local assembly code generated by Firedrake for a Burgers problem on a 3D tetrahedral mesh with Lagrange $p = 1$ elements.

Object name	Type	Variable name(s)
Determinant of Jacobian matrix	double	det
Inverse of Jacobian matrix	double[]	K
Coordinates	double**	coords
Fields	double**	w
Integration points	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. An example is provided in Figure 2, which shows an 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 a non-linear problem, like Burgers 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. Also, constants evaluated in outer loops (called F in the code), acting as scaling factors of arrays, may be required; trip counts can be larger (proportionally to the order of the method); arrays may be block-sparse. Note that in addition to a larger number of operations to compute the element matrix, the Burgers case shows a register pressure higher than that in Helmholtz. Despite assembly kernels being problem-dependent, meaning that the space of codes that Firedrake can generate is infinite, it is still possible to identify common domain-specific traits that can be exploited for effective code transformations and SIMD vectorization.

Some peculiarities of assembly kernels are: 1) memory accesses along the three loop dimensions are always stride-1; 2) the j and k loops are interchangeable, whereas permutation of i might be subjected to pre-computation of values (e.g. the F values in Burgers) and introduction of temporary arrays; 3) the j and k loops 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 III we show how to exploit these observations to define a set of systematic, composable optimizations.

III. Code Transformations

The code transformations presented in this section are applicable to all finite element problems that can be formulated in Firedrake. One peculiar characteristic is that they all aim at improving the run-time of the `ijk` loop nest, where the numerical integration takes place. In rare, yet important cases, however, this might not be sufficient to achieve notable speedups. Generalized loop-invariant code motion, which is described in Section III-B, is a fundamental optimization that allows pre-computation of invariant sub-expressions. It is worth noting there are circumstances in which the amount of lifted terms, either at the level of an outer loop or even completely outside of the loop nest, is so big (thousands of operations, hundreds of temporaries) that their execution time becomes the dominant factor of the overall local assembly run-time. In these cases, two challenging optimizations are common subexpression elimination and effective vectorization (superword level parallelism [13] would not be of help, because invariant sub-expressions lack, in general, structure). Given its inherent complexity, a comprehensive study of this problem is left as further work.

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 iterations, render the optimization process challenging. Relatively to machine peak, while some problems achieve good performance when applying a very specific set of transformations, there is also an ample space of cases for which a more elaborated optimization strategy is required. 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.

A. 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 smaller than 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 that the underlying machine possesses AVX, which implies $vl = 4$ since a vector register is 256 bits long and our kernels use 64-bits double-precision floating-point values. In this example, 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 of a remainder (scalar) loop from the compiler, which would render vectorization less efficient.

B. Generalized Loop-invariant Code Motion

From inspection of the codes in Figures 2 and 3, it can be noticed that the computation of A involves evaluating many sub-expressions that depend on two iteration variables only. 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 have. First, they can identify sub-expressions that are invariant with respect to the innermost loop only. 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, because 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 be otherwise significant. It is also worth noting that, in some problems, invariant sub-expressions along j are identical to those along k (e.g. in Helmholtz). In these cases, we safely avoid redundant pre-computation since, as anticipated in Section II, 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.

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

  static const double W3[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*W3[i]);
  }
}
```

Fig. 4. Local assembly code generated by Firedrake when padding, data alignment, and $licm$ are applied to the Helmholtz problem given in Figure 2. Data alignment and padding are for an AVX machine. In this specific case, sub-expressions invariant to j are identical to those invariant to k, so they can be precomputed once in a single loop r; in general, this might not be the case.

C. 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 accessed variables are such that the vector-registers pressure is high. Since the kernel's working set fits the L1 cache, it is remarkably important to optimize register management. Canonical 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]`, we can notice that the computation of the element matrix is abstractly expressible as

$$A_{jk} = \sum_{\substack{x \in B' \subseteq B \\ y \in B'' \subseteq B}} x_j \cdot y_k \qquad j,k = 0,...,4 \tag{1}$$

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, and by abstracting from the presence of all variables independent of both j and k, the element matrix computation is always reducible to this form. Figure 5 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. Besides 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

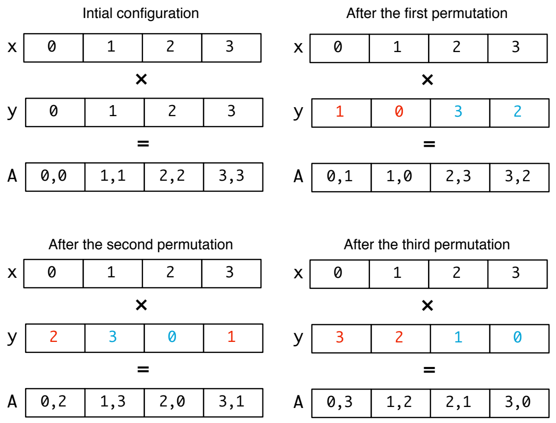

Fig. 5. Outer-product vectorization by permuting values in a vector register.

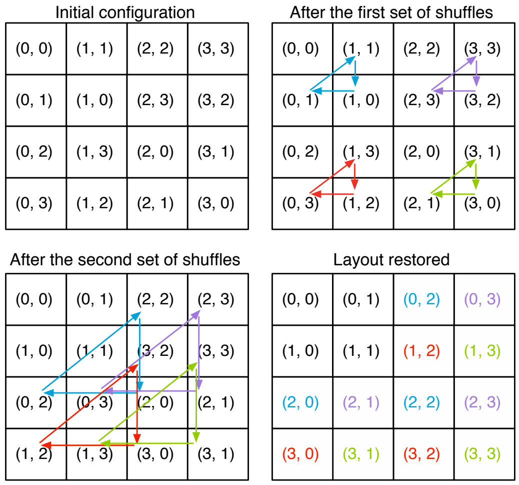

Fig. 6. Restoring the storage layout after *op-vect*. The figure shows how 4×4 elements in the top-left block of the element matrix A can be moved to their correct positions. Each rotation, represented by a group of three same-colored arrows, is implemented by a single vector shuffle intrinsics.

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 6, 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 7.

D. 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, it is likely to obtain poor register reuse. 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 needs 33 temporaries for the `ijk` loop order, and compiler's hoisting of invariant loads out of the k loop can be inefficient on architectures with a relatively low number of registers. One potential solution to this problem consists of suitably "splitting" the computation of the element matrix A into multiple sub-expressions; an example, for the Helmholtz problem, is given in Figure 8. Splitting an expression has, in general, several 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 (e.g. the product $det * W3[i]$ is performed times number of sub-expressions in the code of Figure 8). Finally, it might affect register locality, although this is not the case of the Helmholtz example: for instance, the same array could be accessed in different sub-expressions, requiring a proportional number of loads be performed. Nevertheless, as shown in Section V, 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.

```
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) {
        // Call Load and set intrinsics
        // Compute A[1,1],A[2,2],A[3,3],A[4,4]
        // One permute_pd intrinsics per k-loop load
        // Compute A[1,2],A[2,1],A[3,4],A[4,3]
        // One permute2f128_pd intrinsics per k-loop load
        // ...
      }
    // Do 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]
    }
  }
}
```

Fig. 7. Local assembly code generated by Firedrake when padding, data alignment, *licm* and *op-vect* are applied to the Helmholtz problem given in Figure 2. 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.

Table II summarizes the code transformations described so far. Given that many of these transformations depend on

```
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];
}
```

Fig. 8. Local assembly code generated by Firedrake when *split* is applied to the optimized Helmholtz problem given in Figure 4. In this example, the split factor is 2.

some parameters (e.g. tile size), we need a mechanism to prune such a large space of optimization. This aspect is treated in Section IV.

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.

IV. OVERVIEW OF COFFEE

Firedrake provides users with the Unified Form Language to write problems in a notation resembling mathematical equations. At run-time, the high-level specification is translated by the Fenics Form Compiler [9] 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 III. The output of COFFEE, C code, is eventually provided to PyOP2 [15], where just-in-time compilation and execution over the discretized domain take place. Because of the large number of (possibly 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, an 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 non-affine loop over mesh elements is vectorized, can be applied. This feature, currently under development, has been proved to be useful in several finite volume applications [21]. The second transformation step is applied if the backend is a manycore architecture, like a GPU: the compiler tries to extract parallelism from inside the kernel, by partitioning loop iterations among different threads, if these are found to be independent [15]. Then, 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 proper AVX or LRBNi 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 reminder loop, amenable to auto-vectorization, is inserted.

```
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()
endif
if isa.backend == gpu then
    if it_space then
        ast.extract_iteration_space(wrapper)
    endif
    return wrapper + ast.from_ast_to_c()
endif
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()
endif
if plan.sz_split then
    ast.split(plan.sz_split)
endif
if plan.uaj_factor then
    uaj = MIN(plan.uaj_factor, ⌊it_space.j.size/isa.vf⌋)
    ast.op_vect(uaj)
endif
return wrapper + ast.from_ast_to_c()
```

Fig. 9. Pseudocode of the COFFEE pipeline.

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 for every iteration. Experiments showed that the performance loss is greater than the gain due to the possibility of accumulating increments in a register, rather than in 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 loop along which the number of invariant loads is smaller as outer so that more registers remain available to carry out the computation of the element matrix.

Loop unroll and unroll-and-jam of outer loops are fundamental to expose instruction-level parallelism, so tuning critical parameters, like the unroll factor, is of great importance. However, from our experience of inspecting assembly code and comparing with other hand-written implementations, recent versions of a vendor compiler like Intel's employ cost models capable of estimating close-to-optimal values for such parameters. This is particularly true for assembly kernels, where the loop nest is affine, bounds are usually very small and known at compile-time, and memory accesses are unit-stride. We therefore leave the backend compiler in charge of selecting the unroll factor. This choice also simplifies the 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. Note that, regardless of the output of the cost model, the unroll-and-jam factor never exceeds the actual size of the outer loop (line 25 in Figure 7, assuming the default loop order `ijk`).

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).

V. Performance Evaluation

A. Experimental Setup

Experiments were run on two Intel machines, a Sandy Bridge (I7-2600 CPU, running at 3.4GHz, 32KB L1 cache and 256KB L2 cache) and the Xeon Phi. The `icc 13.1` compiler was used, with optimization level `-O2` and with auto-vectorization enabled (`-xAVX` on the Sandy Bridge, and `TODO` on the Phi). Other optimization levels performed, in general, slightly worse than `-O2`. Our code transformations were evaluated in three real-world problems based on the following PDEs:

- Helmholtz
- Advection-Diffusion
- Burgers

The code was written in UFL 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 (Figure 3), where a larger number of basis function matrices ($X1$, $X2$, ...) and constants ($F0$, $F1$, ..., $K0$, $K1$, ...) are accessed.

```
1  **Cost Model**
   **Input**: n_outer_arrays, n_inner_arrays, n_consts, n_regs
   **Output**: uaj_factor, split_factor
2  n_outer_regs = n_regs / 2
3  split_factor = 0
4  // Compute spltting factor
5  while n_outer_arrays > n_outer_regs do
6      n_outer_arrays = n_outer_arrays / 2
7      split_factor = split_factor + 1
8  endw
9  // Compute unroll-and-jam factor for op-vect
10 n_regs_avail = n_regs - (n_outer_arrays + n_consts)
11 uaj_factor = $\lceil$n_reg_avail / n_inner_arrays$\rceil$
12 if n_outer_arrays > n_inner_arrays then
13     permute = True
14 else
15     permute = False
16 endif
17 return <permute, split_factor, uaj_factor>
```

Fig. 10. 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.

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. Intuitively, the bigger the element shape and p, the larger is the iteration space. Triangles (2D), tetrahedron (3D), and prisms (3D) 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 kind of elements is, respectively, 3, 4, and 6. Moving from 2D to 3D has the effect of increasing the number of basis function arrays, since, conceptually, the behaviour 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 such a large space of problems, the size of the element matrix rarely exceeds 30×30, with a peak of 105×105 in Burgers with prisms and $p = 4$.

B. Impact of Generalized Loop-invariant Code Motion

Table V-B illustrates the speed-ups obtained on the Sandy Bridge and the Phi machines when *licm*, data alignment, and padding are used, over non-transformed code. We distinguish between *licm* and *licm-ap*; the latter makes use of padding and data alignment. Inspection of assembly code generated

		AVX		LRBNi	
		licm	licm+ap	licm	licm+ap
Helmholtz	triangle	1.05×-1.68×	1.32×-4.14×	1.34×-4.32×	1.34×-4.32×
Helmholtz	tetrahedron	1.36×-2.64×	1.35×-3.32×	1.34×-4.32×	1.34×-4.32×
Helmholtz	prism	2.16×-2.45×	2.42×-2.63×	1.34×-4.32×	1.34×-4.32×
Diffusion	triangle	1.09×-1.64×	1.37×-4.28×	1.34×-4.32×	1.34×-4.32×
Diffusion	tetrahedron	1.30×-3.11×	1.41×-3.82×	1.34×-4.32×	1.34×-4.32×
Diffusion	prism	1.81×-2.70×	2.55×-3.13×	1.34×-4.32×	1.34×-4.32×
Burgers	triangle	1.53×-2.46×	1.56×-2.77×	1.34×-4.32×	1.34×-4.32×
Burgers	tetrahedron	1.58×-2.24×	1.59×-2.10×	1.34×-4.32×	1.34×-4.32×
Burgers	prism	1.32×-2.11×	1.42×-2.31×	1.34×-4.32×	1.34×-4.32×

TABLE III. IMPACT OF GENERALIZED LOOP-INVARIANT CODE MOTION (*licm* COLUMN) ON THE HELMHOLTZ, DIFFUSION AND BURGERS PROBLEMS, FOR THREE KIND OF ELEMENTS BELONGING TO THE LAGRANGE FAMILY (TRIANGLE, TETRAHEDRON, PRISM), FOR THE RANGE OF POLYNOMIAL ORDERS $p \in \{1, 4\}$. EACH ENTRY INDICATES THE RANGE OF SPEED-UPS OBTAINED OVER THE NON-OPTIMIZED IMPLEMENTATION. THE COLUMN *licm-ap* ILLUSTRATES THE COMBINATION OF *licm* WITH DATA ALIGNMENT AND PADDING. RESULTS ARE SHOWN FOR BOTH THE SANDY BRIDGE AND THE XEON PHI ARCHITECTURES.

by icc confirmed all limitations described in Section III-B: only sub-expressions invariant with respect to outer loops are hoisted and, interestingly, not vectorized. This motivates the usually significant gain obtained. Padding and data alignment enhance, in general, the quality of SIMD auto-vectorization. Sometimes the run-time of *licm-ap* is similar to that of *licm* because the element matrix size is, without padding, already a multiple of the vector length, and data is automatically aligned. Occasionally *licm-ap* is slower than *licm* (e.g. in Burgers 3D p3). This is due to the large number of aligned temporaries introduced by *licm*, which probably induces cache associativity conflicts.

C. Impact of Vector-register Tiling

Figures 11 and 12 show the speed-ups achieved applying *op-vect* on top of *licm-ap* to the Helmoltz and Diffusion kernels, respectively. 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 the COFFEE's cost model. In general, there is no substantial difference between the two. This is chiefly because assembly kernels fit the L1 cache, so estimating the run-time based on the quality of register allocation is relatively simple. The rationale behind these results is that, for smaller configurations, vector-register tiling is rarely helpful. This is due to the fact that in the considered problems 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 minimizes pipeline stalls, leading to maximum speed-ups of 1.4× in Diffusion and of 1.24× in Helmholtz on the Sandy Bridge architecture. This was proved using the Intel Architecture Code Analyzer tool [1]. In Helmholtz, 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 consequence of increased register re-use, which relieves the pressure on both data (cache) ports available in the processor.

Application of *op-vect* to the Burgers problem induces meaningful slow downs, due to the large number of temporary arrays that need to be tiled, which exceeds the number of available logical registers on the underlying machine. Expression splitting can be used in combination to *op-vect* to alleviate this issue; this is discussed in Section V-D.

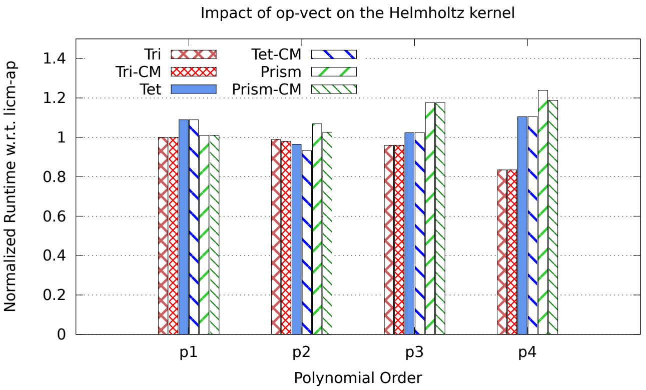

Fig. 11. Speed-ups obtained by applying *op-vect* on top of *licm-ap* to the Helmholtz kernel. Tri, Tet, and Prism stand, respectively, for triangual, tetrahedral and prismatic mesh. CM stands for cost model.

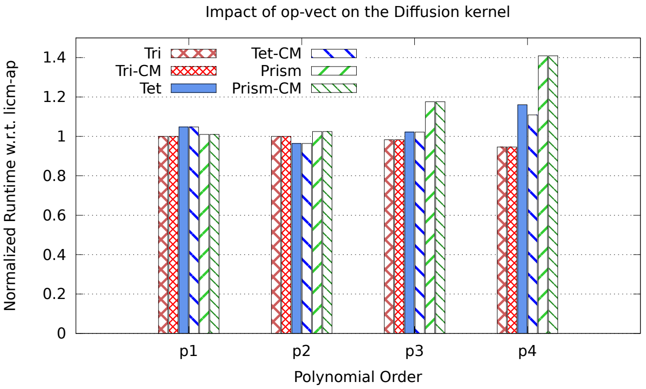

Fig. 12. Speed-ups obtained by applying *op-vect* on top of *licm-ap* to the Diffusion kernel. Tri, Tet, and Prism stand, respectively, for triangual, tetrahedral and prismatic mesh. CM stands for cost model.

D. Impact of Expression Splitting

Expression splitting, or loop fission, contributes to relieve register pressure. The transformation is useless if the element matrix can be evaluated by accessing a relatively small number of basis functions (or temporary values introduced by *licm*), for which efficient register allocation can be already guaranteed. In the cases of Helmholtz and Diffusion, irrespective of split factor and cut, the only consequence of applying *split* is to augment the number of increments for evaluating the element matrix. On the Sandy Bridge machine, slow downs up to 1.4× were observed in these two problems. The cost model, however, prevents the adoption of this transformation, since the `while` statement at line 5 in Figure 8 is never entered. In the Burgers problem, *split* allows notable speed-ups to be achieved over the best run-time provided by either *licm* or *licm-ap*. Results are shown in Figure 13. In almost all problem configurations, a split factor of 1 induced optimal performance, meaning that the original expression was divided into two parts, with one part performing a slightly larger number of floating point operations than the other. Register locality could have been affected by requiring the same array be re-loaded twice as before splitting. However, in this experiment, this was only partly true, because most of the arrays appear in one sub-expression only.

The *split* transformation was also tried in combination with *op-vect* (*split-op-vect*), although the cost model prevents the application of the latter on the Sandy Bridge. Despite performance improvements up to 1.22×, *split-op-vect* never outperforms *split*. This is because of two reasons: for small split factors, like 1 and 2, the data space to be tiled is still too big, and register spilling still affects the run-time; for higher ones, sub-expressions become so small that, as explained in Section V-C, full or partial unrolling of the loop nest allows achieving a certain degree of register re-use.

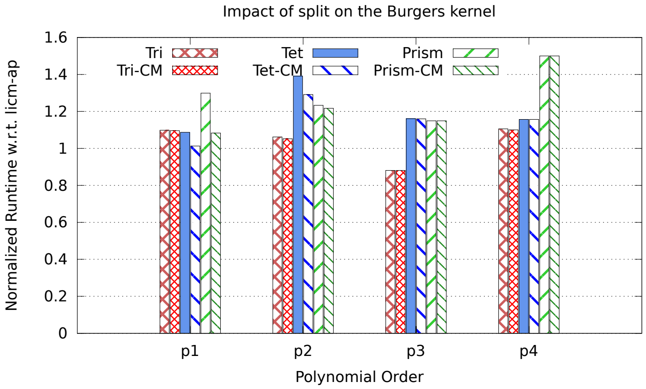

Fig. 13. Speed-ups obtained by applying *op-vect* on top of *licm-ap* to the Burgers kernel. Tri, Tet, and Prism stand, respectively, for triangual, tetrahedral and prismatic mesh. CM stands for cost model.

E. Comparison with Fenics Form Compiler's built-in Optimizations

We have modified the Fenics Form Compiler (FFC) to make it 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 [14]. In FEniCS, FFC can apply its own model-driven optimizations to local assembly kernels [18], which mainly consist of loop-invariant code motion and elimination, at code generation time, of floating point operations involving zero-valued columns in basis function arrays. The FEniCS' loop-invariant code motion is, however, definitely different from COFFEE's, since it is based on expansion of arithmetic operations to identify terms that are invariant of the whole loop nest. Thus, depending on the way expansion is performed, operation count does not necessarily decrease. On the other hand, elimination of zero-valued columns, which are due to certain properties of the input 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 affect vectorization because of the need for gather operations. The price to pay is that only some of the zero-columns will be removed.

Table V-E summarizes the results of the comparison between FEniCS' (FFC) optimizations and COFFEE on the Sandy Bridge. In COFFEE, 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, $p = 1$), although they are generally milder than the maximum one reported in table, obtained with a triangular mesh and $p = 1$. This is a result of zero-columns removal in FEniCS: 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 even in these cases as well. In the cases of Helmholtz and Diffusion, the minimum speed-ups are, respectively, 1.10× and 1.18× (2D mesh, $p = 1$), which keep increasing with polynomial order and element shape up to the values illustrated in the table.

Problem	Max slow down	Max speed up
Helmholtz	/	11.72×
Diffusion	/	14.68×
Burgers	2.24×	1.61×

TABLE IV. SUMMARY OF PERFORMANCE COMPARISON BETWEEN FENICS AND COFFEE ON THE SANDY BRIDGE MACHINE. THE SLOW DOWN AND SPEED UP COLUMNS COMPARE THE BEST COFFEE IMPLEMENTATION AND THE FENICS IMPLEMENTATION WITH OPTIMIZATIONS ENABLED.

F. Comparison with hand-made BLAS-based implementations

For the Helmholtz 3D problem, 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 only *licm*, data alignment and padding were used, with the slow down being even worse for smaller problem sizes. These experiments justify that there is a set of problems for which turning to MKL is not beneficial in terms of performance. It is possible that employing certain BLAS implementations or auto-tuning strategies for particular problem sizes is helpful, although the same problems afflicting *split* (exposed in Section III-D) might limit the performance gain. However, as anticipated, extraction of matrix-matrix multiplies from analysis of the kernel's AST and a strategy for their execution will be comprehensively addressed in further work.

VI. Related Work

The finite element method is used in the most disparate contexts to approximate solutions of PDEs. Well-known frameworks and applications include nek5000 [19], the Fenics project [14], Fluidity [4], and of course Firedrake; this is not an exhaustive list, though. 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 [10] and symbolic manipulation [23] have been developed, although it has been demonstrated that numerical integration remains the optimal choice for a wide class of problems [18].

Optimization of local assembly using numerical integration for CPU platforms has been tackled in [18]. However, to the best of our knowledge, ours is the first work targeting low-level optimizations and adopting a real compiler approach. In [16], and more recently in [11], the problem has been studied for GPU architectures. In [12], variants of the standard numerical integration algorithm have been specialized for the PowerXCell processor and evaluated; the paper lacks, however, an exhaustive study from the compiler viewpoint as we did, and none of the optimizations presented in Section III are mentioned.

Domain-specific languages and automated code generation have been adopted with success in different areas. Spiral [20] automates generation of highly-optimized platform-specific digital signal processing numerical algorithms. Similar ideas are adopted in [25], where a DSL and a compiler for dense linear algebra kernels are proposed. OP2 [17], and its python implementation [15], which is used by Firedrake to express iteration over meshes, aim at achieving performance portability for scientific codes based on unstructured meshes. Stencil DSLs and compilers, such as Pochoir [27] and SDSL [8], have been developed to support development and high performance in fields like image processing and scientific computations over structured grids.

COFFEE currently uses a simple cost model and heuristics to steer the optimization process. Many code generators, like those based on the Polyhedral model [5] and those driven by domain-knowledge [26], make use of cost models. The opposite extreme consists of relying on auto-tuning to select the best implementation for a given problem on a certain platform. Notable examples include tuning of small matrix-matrix multiplies in nek5000 [24], the ATLAS library [28], and FFTW [7] for fast fourier transforms. In both cases, pruning the implementation space, as we did in Section IV, is fundamental to mitigate complexity and overhead.

VII. 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, this is the first compiler oriented towards the introduction of 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 matrices and scalar, 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 way 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 the kernel into a sequence of BLAS calls. COFFEE supports all of the problems expressible in Firedrake, and it is already integrated with this framework.

Acknowledgment

This research is partly funded by the MAPDES project and by the Department of Computing at Imperial College London. The authors would like to thank Dr. Lawrence Mitchell and Dr. Francis Russell for their invaluable suggestions and their contribution to the Firedrake project.

References

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

[2] The firedrake project, 2014.

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

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

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

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

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

[8] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA, 2013. ACM.

[9] Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32(3):417–444, September 2006.

[10] Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32(3):417–444, September 2006.

[11] Matthew G. Knepley and Andy R. Terrel. Finite element integration on gpus. *ACM Trans. Math. Softw.*, 39(2):10:1–10:13, February 2013.

[12] Filip Kruel and Krzysztof Bana. Vectorized opencl implementation of numerical integration for higher order finite elements. *Comput. Math. Appl.*, 66(10):2030–2044, December 2013.

[13] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM.

[14] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

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

[16] Graham R. Markall, David A. Ham, and Paul H.J. Kelly. Towards generating optimised finite element solvers for {GPUs} from high-level specifications. *Procedia Computer Science*, 1(1):1815 – 1823, 2010. {ICCS} 2010.

[17] G.R. Mudalige, M.B. Giles, J. Thiyagalingam, I.Z. Reguly, C. Bertolli, P.H.J. Kelly, and A.E. Trefethen. Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems. *Parallel Computing*, 39(11):669 – 692, 2013.

[18] Kristian B. Olgaard and Garth N. Wells. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Softw.*, 37(1):8:1–8:23, January 2010.

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

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

[21] I. Z. Reguly, E. László, G. R. Mudalige, and M. B. Giles. Vectorizing unstructured mesh computations for many-core architectures. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'14, pages 39:39–39:50, New York, NY, USA, 2007. ACM.

[22] Istvan Z. Reguly, Gihan R. Mudalige, Carlo Bertolli, Michael B. Giles, Adam Betts, Paul H. J. Kelly, and David Radford. Acceleration of a full-scale industrial cfd application with op2. http://arxiv.org/abs/1403.7209v1, March 2014.

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

[24] Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul D. Hovland. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 253–262, New York, NY, USA, 2010. ACM.

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

[26] Kevin Stock, Tom Henretty, Iyyappa Murugandi, P. Sadayappan, and Robert Harrison. Model-driven simd code generation for a multiresolution tensor kernel. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1058–1067, Washington, DC, USA, 2011. IEEE Computer Society.

[27] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[28] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.