A Domain-Specific Optimizing Compiler for Solving Partial Differential Equations

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

***Abstract*—The abstract goes here.**

I. Introduction

In many fields, like computational fluid dynamics, computational electromagnetics, and structural mechanics, programs model phenomena by means of partial differential equations (PDEs). Numerical techniques, like Finite Volume Method and Finite Element Method, are widely employed in real-world applications 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 cell of the discretized domain by applying suitable numerical kernels. As the number of cells 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 [3], demonstrates the effectiveness of the DSL approach for implementing PDEs solvers [**?**].

OP2 adopts a kernel-oriented programming model, in which the computation semantics is expressed through self-contained functions ("kernels"). A kernel is applied to all elements in a set of mesh components (e.g. edges, vertices, elements), 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 by [**?**], [**?**], [2]. Kernel optimization is one of the major concerns in unstructured mesh applications. In this paper, we tackle this problem by proposing a domain-driven 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 runtime, typically 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, usually non-perfect loop nest, where individual loops are rather small (the trip count rarely exceeds 30, with a minimum value of 3, depending on the order of the method). With such small kernels, we focus on aspects like minimization of floating-point operations, register allocation and instruction-level parallelism, especially in the form of SIMD vectorization. Our study is conducted in the context of Firedrake, a system for solving PDEs through the Finite Element Method based on the OP2 abstraction [**?**].

Optimization of assembly kernels is non-trivial. Polyhedral compilation could be used to improve performance; however, given the structure and the exceptionally small size of the kernels, effective SIMD vectorization can be obtained only through an in-depth analysis of both data and iteration space, which is not supported by state-of-the-art polyhedral compilers. BLAS routines could be employed as well, although fairly complicated control- and data-flow analysis would be required to automate identification and extraction of matrix-matrix multiplications. BLAS libraries are also known to perform distant from peak performance when the dimension of the matrices is small [5]. As detailed in Section V, hand-made BLAS implementations of the Helmholtz assembly kernel (illustrated later) have run-times worse than those achieved with our optimization strategy.

Given the constraints on polyhedral compilation and linear-algebra-specialized libraries, we address assembly optimization by studying a set of domain-specific code transformations, applicable to a wide class of problems. We have developed an optimizing compiler and we have integrated it with Firedrake. This allows us to evaluate our code transformations in a range of real-world problems, varying parameters that impact both solution accuracy and kernel cost, namely the polynomial order of the method (from $p = 1$ to $p = 4$) and the mesh type (2D, 3D, 3D hybrid structured-unstructured).

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 33% over "softly-optimized" code (i.e. where only basic transformations are performed, such as auto-vectorizable loop-invariant code motion, padding and data alignment), and up to 77% over original kernels. The contribution of this paper is

```
1  void helmholtz(double A[3][3], double **coords) {
2    // K, det = Compute Jacobian (coords)
3
4    static const double W3[3] = {...}
5    static const double X_D10[3][3] = {{...}}
6    static const double X_D01[3][3] = {{...}}
7
8    for (int i = 0; i<3; i++)
9      for (int j = 0; j<3; j++)
10       for (int k = 0; k<3; k++)
11         A[j][k] += ((Y[i][k]*Y[i][j]+
12           +((K1*X_D10[i][k]+K3*X_D01[i][k])*
13           *(K1*X_D10[i][j]+K3*X_D01[i][j]))+
14           +((K0*X_D10[i][k]+K2*X_D01[i][k])*
15           *(K0*X_D10[i][j]+K2*X_D01[i][j])))*
16           *det*W3[i]);
17 }
```

Algorithm 1: Local assembly code generated by Firedrake for a Helmholtz problem (2D mesh, Lagrange $p = 1$ elements).

therefore twofold:

- An optimisation strategy for a class of kernels widely-used in scientific applications, namely Local Assembly in the context of the Finite Element Method. Our approach exploits domain knowledge to go beyond the limits of both vendor (e.g. icc) and research (e.g. polyhedral) compilers.
- Automation of such code optimizations in Firedrake. This enables performing an in-depth performance evaluation of the proposed optimization strategy in real-world simulations.

The paper is organized as follows. ...

II. Background

Local assembly consists of evaluating so called element stiffness matrix and element stiffness vector; in this work, we focus on computation of matrices, which is the costly part of the process. A stiffness matrix can be intuitively thought as an approximated representation of the PDE solution in a specific cell (or element) of the discretized domain. Numerical integration algorithms are widely-used in assembly codes to implement evaluation of stiffness matrices [4], [**?**].

Given a mathematical description of the input problem, expressed through the domain-specific Unified Form Language [**?**], 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. In our work, we have enhanced 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. Notable examples are [**?**], [**?**], [**?**], in which the kernel's loop nest and memory access pattern are reducible to that generated by Firedrake.

The complexity of Firedrake-generated kernels depends on the mathematical problem being solved. In simpler cases, the loop nest is perfect, it has short trip counts (in the range

```
1  void burgers(double A[12][12], double **c, double **w) {
2    // K, det = Compute Jacobian (c)
3
4    static const double W5[5] = {...}
5    static const double X1_D001[5][12] = {{...}}
6    static const double X2_D001[5][12] = {{...}}
7    //It follows 11 other basis functions definitions.
8    ...
9
10   for (int i = 0; i<5; i++) {
11     double F0 = 0.0;
12     //It follows 10 other declarations (F1, F2,...)
13     ...
14     for (int r = 0; r<12; r++) {
15       F0 += (w[r][0]*X1_D100[i][r]);
16       //It follows 10 analogous statements (F1, F2, ...)
17     }
18     ...
19     for (int j = 0; j<12; j++)
20       for (int k = 0; k<12; k++)
21         A[j][k] += (..(K5*F9)+(K8*F10))*Y1[i][j])+
22           +(((K0*X1_D100[i][k])+(K3*X1_D010[i][k])+
23           +(K6*X1_D001[i][k]))*Y2[i][j]))*F11)+
24           +(..((K2*X2_D100[i][k])+...+(K8*X2_D001[i][k])*
25           *(K2*X2_D100[i][j])+...+(K8*X2_D001[i][j]))..)+
26           + <roughly a hundred of sum/muls go here>)..)*
27           *det*W5[i]);
28   }
29 }
```

Algorithm 2: Local assembly code generated by FFC for a Burgers problem (3D mesh, Lagrange $p = 1$ elements).

3-15), and the computation reduces to a summation of a few products. An example is provided in Figure 1, which shows an assembly kernel for a Helmholtz problem, using Lagrange basis functions on 2D elements with polynomial order $p = 1$. The stiffness matrix in the code is called A. In other scenarios, for instance when solving a non-linear problem like Burgers (see Figure 2), the number of arrays involved in the computation of A 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 stiffness matrix, the Burgers case shows a register pressure higher than that in Helmholtz. Despite assembly kernels being problem-dependent, meaning that infinite is the space of codes that Firedrake can generate, it is still possible to identify common domain-specific traits, which can be exploited for effective code transformations and SIMD vectorization.

The class of kernels we are considering has, in particular, some peculiarities. 1) The computation of the Jacobian, which is the first step of the assembly, is independent of the loop nest. This is not true in general, since the unstructured mesh might use bended elements that would require the Jacobian be re-computed every i iteration; 2) memory accesses along the three loop dimensions are always 1-stride; 3) 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; 4) the j and k

```
1  void helmholtz(double A[3][4], double **coords) {
2    #define ALIGN __attribute__((aligned(32)))
3    // K, det = Compute Jacobian (coords)
4
5    static const double W3[3] ALIGN = {...}
6    static const double X_D10[3][4] ALIGN = {{...}}
7    static const double X_D01[3][4] ALIGN = {{...}}
8
9    for (int i = 0; i<3; i++) {
10     double LI_0[4];
11     double LI_1[4];
12     for (int r = 0; r<4; r++) {
13       LI_0[r] = ((K1*X_D10[i][r])+(K3*X_D01[i][r]));
14       LI_1[r] = ((K0*X_D10[i][r])+(K2*X_D01[i][r]));
15     }
16     for (int j = 0; j<3; j++)
17       #pragma vector aligned
18       for (int k = 0; k<4; k++)
19         A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j]+
20                     +LI_1[k]*LI_1[j])*det*W3[i]);
21   }
22 }
```

Algorithm 3: Local assembly code generated by Firedrake for a Helmholtz problem (2D mesh, Lagrange $p = 1$ elements). The padding, data alignment and $licm$ optimizations are applied. Data alignment and padding relate to an AVX machine. In this specific case, sub-expressions invariant to j are identical to those invariant to k; in general, this could not be the case.

loops iterate over the same iteration space; 5) so called basis functions arrays (denoted by X, Y, ...) are constants, and most of the sub-expressions on the right hand side of the stiffness 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

A. Padding and Data Alignment

Auto-vectorization of assembly code computing the stiffness 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, both arrays and matrices are allocated to addresses that are multiples of vl. Then, matrices are padded by rounding the number of columns to the nearest multiple of vl. For example, assume the original size of a matrix 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. Then, a padded matrix on this architecture will have size 3×4. The compiler is explicitly informed about data alignment using a suitable pragma. Padding of all matrices involved in the evaluation of the stiffness matrix 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 be responsible for inefficient vectorization.

B. Domain-driven Loop-invariant Code Motion

From inspection of the codes in Figures 1 and 2, 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 icc and gcc have. First, these compilers 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. 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 the 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 3 shows the Helmholtz assembly code after the application of loop-invariant code motion, padding, and data alignment.

C. Domain-drive Optimization of Register Management

One notable problem assembly kernels are exposed to 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. Due to a kernel's working set fitting the L1 cache, it is remarkably important to optimize register management. Canonical optimizations, such as loop interchange, unroll, unroll-and-jam and register tiling are typically employed to deal with this problem. In the compiler we have developed, we support these optimizations either by means of explicit source code transformation (interchange, unroll-and-jam) or indirectly advising the compiler through standard pragmas (unroll). In this context, register tiling is a critical optimization. A way of obtaining register tiles consists of slicing the innermost loop into rectangular blocks of identical size (except the "reminder" block), and applying unrolling to individual blocks in order to expose ILP. This is automated by some vendor compilers [**?**]. Register tiles along both inner dimensions can be determined as well. Based on the observation that the evaluation of the stiffness matrix can be reduced to a "summation of outer products" along the j and k dimensions, we implement a domain-specific vector-register tiling strategy that leads to larger tiles, enhancing reuse. If we consider the code snippet in Figure 3 (Helmholtz after loop-invariant code motion), we can notice that the computation of A 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 matrices (or temporaries) accessed in the kernel, whereas B' and B'' are generic problem-dependent subsets. Without loss of generality, the presence of constants or other variables independent of both j and k can be momentarily neglected. Note that regardless of the specific input problem, the stiffness matrix computation is always reducible to this kind of form. Figure III-C illustrates how we can evaluate 16 elements $(j, k = 0, ..., 4)$ of the stiffness matrix using just 2 vector registers (a 4×4 tile), assuming $|B'| = |B''| = 1$. Values in a register are shuffled each time a product is performed. Standard compiler auto-vectorization (*gcc* and *icc*), 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, and would also need 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.

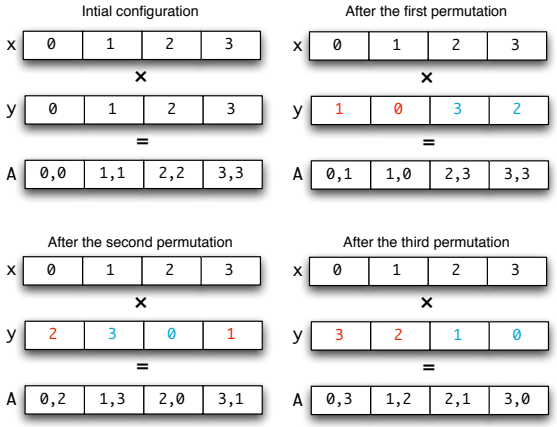

Fig. 1. Outer-product vectorization by permuting values in a vector register.

The storage layout of A, however, is incorrect after the application of this "outer-product vectorization" (*op-vect*). We efficiently restore it with a sequence of vector shuffles following the pattern highlighted in Figure III-C, 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 4.

D. Assembly splitting

In complex kernels, like Burgers in Figure 2, and on certain architectures, achieving effective register allocation can be challenging. If the number of variables independent of the innermost-loop dimension, i.e. basis function matrices, temporaries introduced by loop-invariant code motion and constants, is even close to the number of available CPU registers, it is likely to obtain poor register reuse. 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 stiffness matrix A into multiple sub-expressions; an example, for the simpler Helmholtz problem, is given in Figure 5. Splitting an expression has, in general, several drawbacks. Firstly, it increases the number of

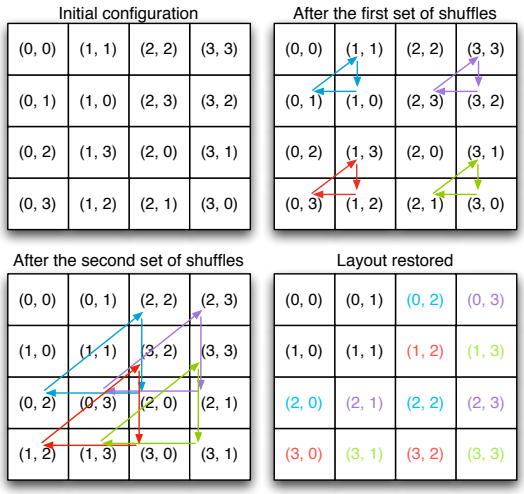

Fig. 2. Restoring the storage layout after *op-vect*. The figure shows how 4×4 elements in the top-left block of the stiffness matrix A can be brought to their correct positions.

accesses to A proportially 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 5). Finally, it might affect register locality, although this is not the case of the Helmholtz example: for instance, the same matrix 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.

Table I 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 IV

IV. The PyOP2 Compiler

Firedrake provides users with the Unified Form Language to write problems in a notation resembling mathematical equations. This high-level specification is translated by the Fenics Form Compiler [**?**] into an Abstract Syntax Tree representation of a Finite Element assembly kernel. ASTs are then passed to PyOP2 [**?**], which lies at the core of Firedrake, where parallel execution over the unstructured mesh is managed. Our compiler, capable of applying the transformations described in Section III, is integrated with PyOP2: it receives FFC's ASTs as input, it introduces optimizations, and it generates C code as output, which is eventually just-in-time compiled on the underlying architecture. Because of the large number of (possibly parametric) transformations, we need a mechanism to select the most suitable optimization strategy for a given problem. Auto-tuning is not an attractive approach since it would be performed at run-time when kernel code is available. The optimization strategy is described in the following, along with an overview of our compiler.

```
1  void helmholtz(double A[8][8], double **coords) {
2    // K, det = Compute Jacobian (coords)
3    // Declaration of basis function matrices
4
5    for (int i = 0; i<6; i++) {
6      // Do loop-invariant code motion
7      for (int j = 0; j<4; j+=4)
8        for (int k = 0; k<8; k+=4) {
9          // Call Load and set intrinsics
10         // Compute A[1,1],A[2,2],A[3,3],A[4,4]
11         // One permute_pd intrinsics per k-loop load
12         // Compute A[1,2],A[2,1],A[3,4],A[4,3]
13         // One permute2f128_pd intrinsics per k-loop load
14         // ...
15       }
16     // Do Remainder loop (from j = 4 to j = 6)
17   }
18   // Restore the storage layout:
19   for (int j = 0; j<4; j+=4) {
20     __m256d r0, r1, r2, r3, r4, r5, r6, r7;
21     for (int k = 0; k<8; k+=4) {
22       r0 = _mm256_load_pd (&A[j+0][k]);
23       // Load A[j+1][k], A[j+2][k], A[j+3][k]
24       r4 = _mm256_unpackhi_pd (r1, r0);
25       r5 = _mm256_unpacklo_pd (r0, r1);
26       r6 = _mm256_unpackhi_pd (r2, r3);
27       r7 = _mm256_unpacklo_pd (r3, r2);
28       r0 = _mm256_permute2f128_pd (r5, r7, 32);
29       r1 = _mm256_permute2f128_pd (r4, r6, 32);
30       r2 = _mm256_permute2f128_pd (r7, r5, 49);
31       r3 = _mm256_permute2f128_pd (r6, r4, 49);
32       _mm256_store_pd (&A[j+0][k], r0);
33       // Store A[j+1][k], A[j+2][k], A[j+3][k]
34     }
35   }
36 }
```

Algorithm 4: Local assembly code generated by Firedrake for a Helmholtz problem (2D mesh, Lagrange $p = 2$ elements). The padding, data alignment, $licm$ and $op\text{-}vect$ optimizations are applied. Data alignment and padding relate to an AVX machine. 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.

The compiler structure is outlined in Figure 6. Initially, an AST is inspected, looking for the presence of iteration spaces and other 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-based applications [**?**]. The second transformation step is applied if the backend is a manycore machine, 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 [**?**]. 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, the $split$ and $op\text{-}vect$ transformations may be introduced; their implementation is based on analysis and trasformation of the AST. The compiler handles any possible corner cases: for example, if $op\text{-}vect$ is to be applied but the size of the iteration space is not a multiple of the vector

```
1  void helmholtz(double A[3][4], double **coords) {
2    #define ALIGN __attribute__((aligned(32)))
3    // K, det = Compute Jacobian (coords)
4    // Declaration of basis function matrices
5
6    for (int i = 0; i<3; i++) {
7      double LI_0[4];
8      double LI_1[4];
9      for (int r = 0; r<4; r++) {
10       LI_0[r] = ((K1*X_D10[i][r])+(K3*X_D01[i][r]));
11       LI_1[r] = ((K0*X_D10[i][r])+(K2*X_D01[i][r]));
12     }
13     for (int j = 0; j<3; j++)
14       #pragma vector aligned
15       for (int k = 0; k<4; k++)
16         A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j])*det*W3[i];
17     for (int j = 0; j<3; j++)
18       #pragma vector aligned
19       for (int k = 0; k<4; k++)
20         A[j][k] += LI_1[k]*LI_1[j]*det*W3[i];
21   }
22 }
```

Algorithm 5: Local assembly code generated by Firedrake for a Helmholtz problem (2D mesh, Lagrange $p = 1$ elements). The padding, data alignment, $licm$ and $split$ optimizations are applied. Data alignment and padding relate to an AVX machine. In this specific case, the split factor is 2.

Name (Abbreviation)	Parameter
Auto-vectorizable Loop-invariant code motion ($licm$)	
Padding	
Data Alignment	
Loop interchange	loops
Loop unrolling	unroll factor
Register tiling	tile size
Outer-product vectorization ($op\text{-}vect$)	tile size
Assembly splitting ($split$)	split point, split factor

TABLE I. OVERVIEW OF CODE TRANSFORMATIONS FOR FIREDRAKE-GENERATED ASSEMBLY KERNELS.

length, then a reminder loop, amenable to auto-vectorization, is inserted.

The cost model is shown in Figure 7. It takes into account the number of available vector registers (n_regs) and the number of variables iterating along the j and k dimensions (n_consts for independent variables, n_outer_arrays for j variables, and n_inner_arrays for k variables, assuming the ijk loop order) to estimate unroll-and-jam and split factors when, respectively, $op\text{-}vect$ and $split$ are used (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. Finally, the unroll-and-jam factor for the $op\text{-}vect$ transformation is determined as a function of "free" registers, i.e. those not used for keeping hoisted terms (line 9-11).

Loop unroll and unroll-and-jam of outer loops are funda-

```
1  The PyOP2 Compiler
   Input: ast, wrapper, isa
   Output: code
2  // Analyze ast and build optimization plan
3  it_space = analyze(ast)
4  if not it_space then
5      ast.apply_inter_kernel_vectorization(wrapper)
       return wrapper + ast.from_ast_to_c()
6  endif
7  if isa.backend == gpu then
8      if it_space then
9          ast.extract_iteration_space(wrapper)
10     endif
11     return wrapper + ast.from_ast_to_c()
12 endif
13 plan = cost_model(it_space.n_inner_arrays, isa.n_regs)
14 // Optimize ast based on plan
15 ast.licm()
16 ast.padding()
17 ast.data_align()
18 if plan.permute then
19     ast.permute_assembly_loops()
20 endif
21 if plan.sz_split then
22     ast.split(plan.sz_split)
23 endif
24 if plan.uaj_factor then
25     ast.op_vect(plan.uaj_factor)
26 endif
27 return wrapper + ast.from_ast_to_c()
```

Algorithm 6: The PyOP2 compiler.

mental to expose ILP and data reuse, and so tuning critical parameters such as the unroll factor becomes of great importance. Autotuning might be used, although we avoid it to minimize the overhead of Firedrake's just-in-time compilation. It is our experience (inspection of assembly code, comparison with other hand-made implementations), however, that for our assembly kernels, where the loop nest is affine, bounds are known at compile-time, and memory accesses are unit-stride, newest versions of a vendor compiler like icc employ cost models capable of estimating close-to-optimal values for such parameters. We therefore leave the backend compiler in charge to select unroll and unroll-and-jam factors. This choice also simplifies the compiler's cost model. The only situation in which we explicitly unroll-and-jam a loop is when $op\text{-}vect$ is used, since the transformed code seems to prevent the icc compiler from applying unroll-based optimizations, even if specific pragmas are added.

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 every iteration; experiments showed that the performance loss is greater than the gain due to accumulating increments in a register 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

```
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 = ⌈n_reg_avail / n_inner_arrays⌉
12 return <split_factor, uaj_factor>
```

Algorithm 7: The cost model is employed by the compiler to estimate the most suitable unroll-and-jam (when $op\text{-}vect$ is used) and split factors, avoiding the overhead of auto-tuning.

outer, the loop along which the number of invariant loads is smaller so that more registers are availabe to carry out the computation of the stiffness matrix.

V. Performance Evaluation

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 Phi. The `icc 2013` 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 expressible with Firedrake:

- Helmholtz
- Advection-Diffusion
- Burgers

The Helmholtz code has already been shown in Figure 1. 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 stiffness matrix A. Burgers is a time-dependent problem (i.e. the assembly is recalculated every time step based on the result of previous iterations) employing differential operators different than those of Helmholtz. The assembly code, partly exposed in Figure 2, is therefore fairly different, with a larger number of basis function matrices ($X1$, $X2$, ...) and constants ($F0$, $F1$, ..., $K0$, $K1$, ...) employed.

These problems were studied varying both the shape of mesh elements and the polynomial order p of the method. Intuitively, the larger the element shape, the bigger is the iteration space. Triangles (2D), tetrahedron (3D), and prisms (3D) were tested. 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 also increases the number of basis function arrays, since conceptually the behaviour of the equation has to be approximated also along the z dimension. 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 $p = 4$, are 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 stiffness matrix rarely excees 20×20, with a peak of 105×105 in Burgers with prisms and $p = 4$.

For the Helmholtz 3D problem, manual implementations based on MKL BLAS were tested on Sandy Bridge. This particular kernel can be easily reduced to a sequence of four matrix-matrix multiplications that can be computed via calls to BLAS `dgemm`. In the case of $p = 4$, where the stiffness 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. As anticipated, extraction of matrix-matrix multiplications from analysis of the kernel's AST or re-design of the Fenics Form Compiler to explicitly expose these operations will be addressed in further work. However, these experiments justify that there's a set of problems for which turning to BLAS is not beneficial in terms of performance. It is possible that employing BLAS is useful for particularly large problems, for instance Burgers on 3D meshes with $p \geq 3$, although the loss in data locality due to re-loading matrices appearing in multiple products might limit the performance gain.

Table **??**

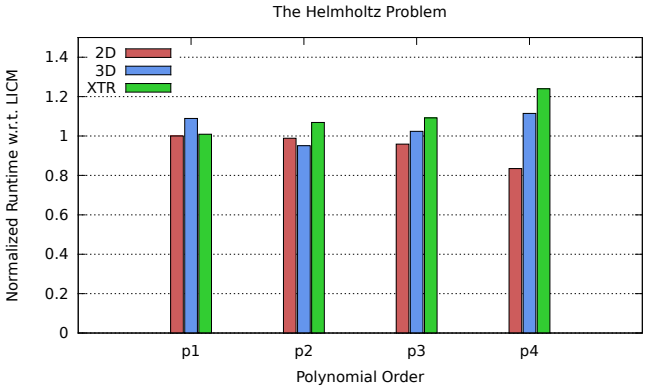

Fig. 3. Helmholtz

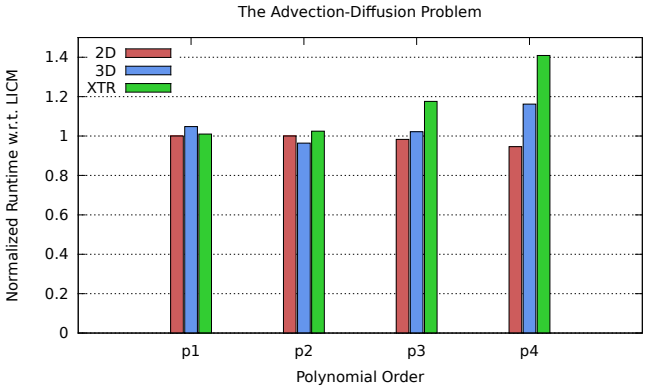

Fig. 4. Advection-Diffusion

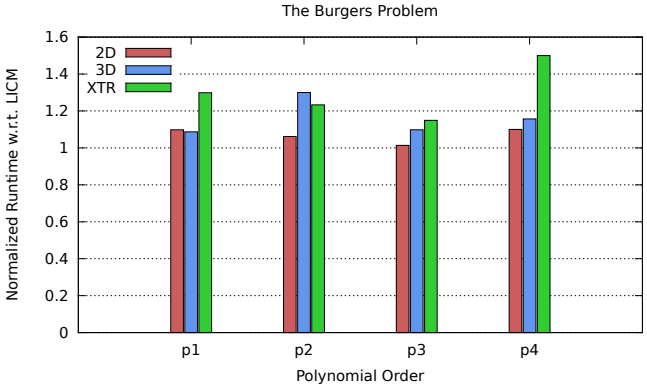

Fig. 5. Burgers

VI. Related Work

VII. Conclusions

Acknowledgment

The authors would like to thank...

References

[1] The firedrake project, 2014.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

				split		*op-vect*	
	original	*licm*	*licm+ap*	Best	Cost Model	Best	Cost Model
Helmholtz_tri_p1		$5.49e{-}17$	$4.95e{-}17$	$1.44e{-}17$	$1.34e{-}17$	$1.56e{-}17$	$\mathbf{4.43e{-}5}$