A

# Optimizing Automated Finite Element Integration through Expression Rewriting and Code Specialization

Fabio Luporini, Imperial College London
David A. Ham, Imperial College London
Paul H.J. Kelly, Imperial College London

The need for rapidly implementing high performance and portable finite element methods has led to approaches based on automated code generation, in which local assembly operations are derived from the mathematical specification of a weak variational form. Local assembly code must be efficient, since the computational cost can dramatically increase with the complexity of the input form, covering a significant fraction of the overall computation run-time. Achieving high performance, however, is non-trivial, due to the complexity of mathematical expressions involved in the numerical integration and the small size of the loop nests, which prevent many standard compiler optimizations. We present an optimization strategy centred on expression rewriting, to minimize the executed floating point operations, and code specialization, to fully exploit the capability of the underlying platform, e.g. SIMD vectorization. By also relying on model-driven, dynamic autotuning, we report notable performance improvements over state-of-the-art optimization systems for local assembly operations.

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

General Terms: Design, Performance

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

**ACM Reference Format:**
Fabio Luporini, David A. Ham, and Paul H. J. Kelly, 2014. Optimizing Automated Finite Element Integration through Expression Rewriting and Code Specialization. *ACM Trans. Arch. & Code Opt.* V, N, Article A (January YYYY), 23 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

## 1. INTRODUCTION

The need for rapidly implementing high performance, robust, and portable finite element methods has led to approaches based on automated code generation. This has been proved successful in the context of the FEniCS [Logg et al. 2012] and Firedrake [Firedrake contributors 2014] projects, which have become incredibly popular over the last years. In these frameworks, the weak variational form of a given problem

This research is partly funded by the MAPDES project, by the Department of Computing at Imperial College London, by EPSRC through grants EP/I00677X/1, EP/I006761/1, and EP/L000407/1, by NERC grants NE/K008951/1 and NE/K006789/1, by the U.S. National Science Foundation through grants 0811457 and 0926687, by the U.S. Army through contract W911NF-10-1-000, and by a HiPEAC collaboration grant. The authors would like to thank Dr. Carlo Bertolli, Dr. Lawrence Mitchell, and Dr. Francis Russell for their invaluable suggestions and their contribution to the Firedrake project.
Author's addresses: Fabio Luporini & Paul H. J. Kelly, Department of Computing, Imperial College London; David A. Ham, Department of Computing and Department of Mathematics, Imperial College London;

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

is expressed at high-level by means of a domain-specific language. Such a mathematical specification is suitably manipulated and then passed as input to a form compiler, whose goal is to generate a representation of local assembly operations. These operations numerically evaluate problem-specific integrals in order to compute so called local matrices and vectors, which represent the contributions from each element in the discretized domain to the equation solution. Local assembly code must be high performance: as the complexity of a variational form increases, in terms of number of derivatives, pre-multiplying functions, and polynomial order of the chosen function spaces, the resulting assembly kernels become more and more computationally expensive, covering a significant fraction of the overall computation run-time.

Producing high performance implementations is, however, non-trivial. The complexity of mathematical expressions involved in the numerical integration, which varies from problem to problem, and the small size of the loop nest in which such integral is computed obstruct the optimization process. Traditional vendor compilers, such as *GNU's* and *Intel's*, fail at exploiting the structure inherent assembly expressions. Polyhedral-model-based source-to-source compilers, for instance [Bondhugula et al. 2008], mainly apply aggressive loop optimizations, such as tiling, but these are not particularly helpful in our context. This lack of suitable optimizing tools has led to the development of a number of higher-level approaches to maximize the performance of local assembly kernels. In [Olgaard and Wells 2010], it is shown how automated code generation can be leveraged to introduce domain-specific optimizations, which a user cannot be expected to write "by hand". [Kirby et al. 2005] and [Russell and Kelly 2013] have studied, instead, different optimization techniques based on a mathematical reformulation of finite element integration. In [Luporini et al. 2014], we have made one step forward by showing that different forms, on different platforms, require distinct sets of transformations if close-to-peak performance must be reached, and that low-level, domain-aware code transformations are essential to maximize instruction-level parallelism and register locality. The problem of optimizing local assembly routines has been tackled recently also for GPU architectures, for instance in [Knepley and Terrel 2013], [Klöckner et al. 2009], and [Bana et al. 2014].

Our research has resulted in the development of a compiler, COFFEE[1], fully integrated with the Firedrake framework [Luporini et al. 2014]. Besides separating the mathematical domain, captured by the form compiler at an higher level of abstraction, from the optimization process, COFFEE also aims to be platform-agnostic. The code transformations occur on an intermediate representation of the assembly operation, which is ultimately translated into platform-specific code. Domain knowledge is exploited in two ways: for simplifying the implementation of code transformations and to make them extremely effective. Domain knowledge is conveyed to COFFEE from the higher level (the form compiler in the case of Firedrake, although any user-provided code would be acceptable) through suitable annotations attached to the input. For example, when the input is in the form of an abstract syntax tree produced by the form compiler, specific nodes are decorated so as to drive the optimization process. Although COFFEE has been thought of as a multi-platform optimizing compiler, our performance evaluation so far has been restricted to standard CPU platforms only. We emphasize once more, however, that the transformations applicable by both the Expression Rewriter (Section 3) and the Code Specializer (Section 4) would work on generic accelerators as well.

In this paper, we build on our previous work [Luporini et al. 2014] and present a novel structured approach to the optimization of automatically-generated finite element integration routines based on quadrature representation. We argue that for com-

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

plex, realistic forms, peak performance can be achieved only by passing through a two-step optimization procedure: 1) expression rewriting, to minimize floating point operations, 2) and code specialization, to ensure effective register utilization and SIMD vectorization. The code transformations introduced in [Luporini et al. 2014] are reused: as explained in Section 2.2, padding and data alignment, expression splitting, and vector-register tiling become steps of code specialization, whereas generalized loop-invariant code motion is performed during the expression rewriting stage. We complement and generalize our previous work with the following contributions.

Expression rewriting aims at reducing the computational intensity of local assembly kernels by rescheduling arithmetic operations based on a set of rewrite rules. Our first contribution consists of a framework that aggressively exploits associativity, distributivity, and commutativity of operators to expose "hidden" loop-invariant sub-expressions and SIMD vectorization opportunities to the code specialization stage. While rewriting an assembly expression, domain knowledge is used in several ways, for example to avoid redundant computation. Secondly, we propose an algorithm that, relying on symbolic execution, restructures the code so as to skip useless iterations over zero-valued entries in vector-valued basis functions arrays, while preserving code vectorizability. These transformations will allow outperforming the results obtained in [Luporini et al. 2014] as well as those achievable using the FEniCS' built-in optimization system, presented in [Olgaard and Wells 2010].

At code specialization time, transformations are applied to maximize the exploitation of the underlying platform's resources, e.g. SIMD lanes. On top of the work in [Luporini et al. 2014], we provide a number of contributions. Firstly, we revisit the padding and data alignment transformation to account for the new code structures produced by expression rewriting. This is of fundamental importance to achieve effective SIMD vectorization. Secondly, we address an open problem in [Luporini et al. 2014] by providing an algorithm that automatically transforms an element matrix evaluation into a sequence of calls to BLAS' dense matrix-matrix multiply routines. BLAS routines are known to perform far from peak performance when the involved arrays are small, which is almost always the case of low-order finite element methods. However, we will show that in corner, yet meaningful cases, especially in forms characterized by the presence of pre-multiplying functions and relatively high-order function spaces, a BLAS-based implementation can be successful. Finally, we introduce a model-driven, dynamic autotuner that transparently evaluates multiple sets of code transformations to determine the best optimization strategy for a given problem. The main challenge here is to build, for a generic form, a reasonably small search space that comprises most of the effective code variants.

Expression rewriting and code specialization have been implemented in COFFEE, which in turn is integrated with the Firedrake framework. Therefore, to testify the goodness of our approach, we provide an extensive and unprecedented performance evaluation across a number of forms of increasing complexity, including some based on complex hyperelasticity models. We characterize our problems by varying polynomial order of the employed function spaces and number of pre-multiplying functions. To clearly distinguish the improvement achieved by this work, we will compare, for each test case, four sets of code variants: 1) unoptimized code, i.e. a local assembly routine as returned from the form compiler; 2) code optimized by FEniCS, i.e. the work in [Olgaard and Wells 2010]; 3) code optimized as described in [Luporini et al. 2014]; code optimized by expression rewriting and code specialization as described in this paper. Notable performance improvements of 4) over 1), 2) and 3) are reported and discussed.

## 2. PRELIMINARIES

### 2.1. Quadrature for Finite Element Local Assembly

We summarize the basic concepts sustaining the finite element method following the notation adopted in [Olgaard and Wells 2010] and [Russell and Kelly 2013]. We consider the weak formulation of a linear variational problem

$$\begin{aligned} &Find\ u\ \in U\ such\ that \\ &a(u,v) = L(v), \forall v \in V \end{aligned} \tag{1}$$

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

$$A\mathbf{u} = b \tag{2}$$

in which $A$ and $b$ discretize $a$ and $L$ respectively:

$$\begin{aligned} A_{ij} &= a(\phi_i(x), \phi_j(x)) \\ b_i &= L(\phi_i(x)) \end{aligned} \tag{3}$$

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

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

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

$$-\nabla \cdot (w \nabla u) = 0 \tag{4}$$

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

$$a(v,u) = \int_\Omega w \nabla v \cdot \nabla u \ \mathrm{d}x \tag{5}$$

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

$$A_{ij}^K = \int_K w \nabla \phi_i^K \cdot \nabla \phi_j^K \ \mathrm{d}x \tag{6}$$

The local element vector $L$ can be determined in an analogous way.

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

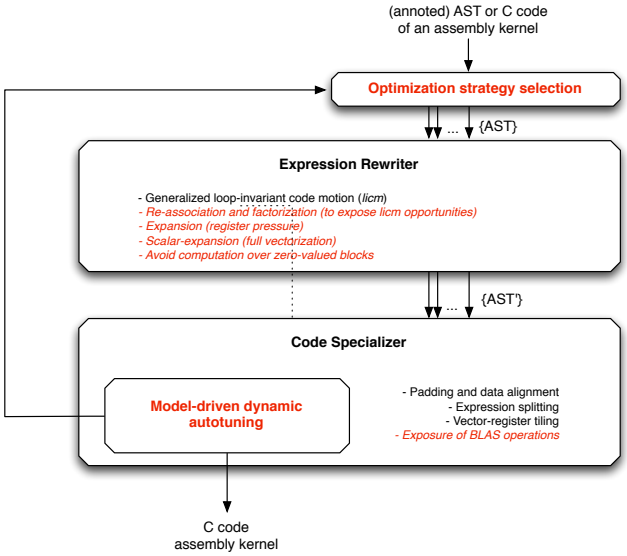


Fig. 1: Outline of COFFEE. The input is either a string of C code or an abstract syntax tree representation. The optimization process goes through two main steps: *expression rewriting* and *code specialization*. The autotuner, incorporated with the Code Specializer, use heuristics to select the set of code variants that, for a given variational form, are likely to reach high performance.

can be expressed as follows

$$A_{ij}^K = \sum_{q=1}^{N} \sum_{\alpha_3=1}^{n} \phi_{\alpha_3}(X^q) w_{\alpha_3} \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial \phi_i^K(X^q)}{\partial X_{\alpha_1}} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \frac{\partial \phi_j^K(X^q)}{\partial X_{\alpha_2}} det F_K' W^q \tag{7}$$

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

In the next sections, we will often refer to the local element matrix evaluation, such as Equation 7 for the weighted Lapalce operator, as the *assembly expression* deriving from the weak variational problem.

### 2.2. COFFEE: a Compiler for Optimizing Quadrature-based Finite Element Integration

If high performance code needs to be generated, assembly expressions must be optimized with regards to three interrelated aspects: 1) arithmetic intensity, 2) instruction-level parallelism, and 3) data locality. In this paper, we tackle these three points building on our previous work ([Luporini et al. 2014]), in which we presented COFFEE, a mature, platform-independent compiler capable of optimizing local assembly code.

The high-level structure of COFFEE is outlined in Figure 1. The input is either C code, explicitly provided by the user as a string, or an abstract syntax tree (henceforth AST) representation automatically generated at the higher level of abstraction. We have integrated COFFEE with the Firedrake framework; here, the input is originated by the FEniCS Form Compiler [Kirby and Logg 2006] in the form of an AST. Decoration of special AST nodes, or equivalently annotation of C code, is extensively used to convey

domain knowledge to COFFEE. This is used to introduce more effective, specialized optimizations, as well as to simplify the implementation.

Two conceptually distinct software modules can be individuated: the Expression Rewriter and the Code Specializer. The former targets arithmetic intensity. Its role is to transform the assembly expression and, therefore, the enclosing loop nest, so as to minimize the number of floating point operations that are performed to evaluate the element matrix (or vector). The latter is tailored to optimizing for instruction-level parallelism, particularly SIMD vectorization, and data (register) locality. As shown in the Figure, the Expression Rewriter manipulates and transforms the original AST, which is then provided to the Code Specializer. As described later, we have invested some effort to ensure that the code transformations applied at expression rewriting time would not break any code specialization opportunity. This, for example, would not be the case if the Expression Rewriter coincided with the FEniCS Form Compiler's built-in optimization system; we will clarify this aspect in Section 3.3.

To neatly distinguish the contributions of this paper from those in [Luporini et al. 2014], in this section we summarize the results of our previous work. Moreover, in Section 5, we will explicitly compare the performance achieved with the transformations presented in this paper to both [Luporini et al. 2014] and [Olgaard and Wells 2010].

In our previous work, we have demonstrated the effectiveness of a set of optimizations for finite element quadrature-based integration routines originated through automated code generation. We have also shown how different subsets of optimizations are needed to achieve close-to-peak performance in different variational forms. In particular, we can summarize our study as follows

— *Generalized Loop-invariant Code Motion*. Compiler's loop-invariant code motion algorithms may not be general enough to optimize assembly expressions, in which different sub-expressions are invariant with respect to more than one loop in the enclosing nest. As reiterated in Section 3, we work around this limitation by performing source-level code motion, while also achieving vectorization of invariant code.
— *Padding and data alignment*. The small size of the loop nest (integration, test, and trial functions loops) require all of the involved arrays to be padded to a multiple of the vector register length so as to maximize the effectiveness of SIMD code. Data alignment can be enforced as a consequence of padding.
— *Vector-register Tiling*. Blocking at the level of vector registers, which we perform exploiting the specific memory access pattern of the assembly expressions (i.e. a domain-aware transformation), improves data locality beyond traditional unroll-and-jam optimizations. This is especially true for relatively high polynomial order (i.e. greater than 2) or when pre-multiplying functions are present.
— *Expression Splitting*. In certain assembly expressions the register pressure is significantly high: when the number of basis functions arrays (or, equivalently, temporaries introduced by loop-invariant code motion) and constants is large, spilling to L1 cache is a consequence for architectures with a relatively low number of logical registers (e.g. 16/32). We exploit sum's associativity to "split" the assembly expression into multiple sub-expressions, which are computed individually.

In Figure 1, we show where these transformations are applied in the COFFEE pipeline; also, the novel contributions of this work are highlighted in red. In the next two sections, we describe the new functionalities of Expression Rewriter and Code Specializer, respectively.

## 3. EXPRESSION REWRITING

Loop-invariant code motion is the key to reduce the computational intensity of an assembly expression. The Expression Rewriter still relies on this technique, although it makes several steps forward.

Firstly, exploiting operators properties like associativity, distributivity, and commutativity, it manipulates assembly expressions to expose more code hoisting and SIMD vectorization opportunities, as well as to balance the register pressure in the various levels of the loop nest. There are many possibilities of rewriting an expression, so the search space can be quite large. We address this problem by defining a simple yet systematic way of rewriting an expression based on a set of formal rewrite rules, which are meticulously applied by COFFEE.

Secondly, the Expression Rewriter re-structures iteration spaces so as to avoid arithmetic operations over tabulated basis functions columns that are statically known to be zero-valued. Zero-valued columns in tabulated basis functions appear, for example, when taking derivatives on a reference element or when using mixed elements. A code transformation eliminating floating point operations on zeros was presented in [Olgaard and Wells 2010]; however, the issue with that is the use of indirection arrays in the generated code, which break many of the optimizations applicable at code specialization time, including SIMD vectorization. In Section 3.3, we present a novel approach based on symbolic execution.

### 3.1. Motivations and Objectives

Consider the local element matrix computation in Figure 2(a), which is an excerpt from a Burgers problem. The assembly expression, produced by the FEniCS Form Compiler, has been deliberately simplified, and code details have been omitted for brevity and readability. In practice, assembly expressions can be much more complex, for example depending on the differential operators employed in the variational form; however, this example is representative enough for highlighting patterns that are common in a large class of problems.

A first glimpse of the code suggests that the sub-expression `a*f0*A[i][j]+b*f1*B[i][j]` is invariant with respect to the innermost (trial functions) loop `k`, so it can be hoisted at the level of the outer loop `j` to avoid redundant computation. This is indeed a standard compiler transformation, supported by any available compilers, so, in principle, there should be no need to transform the source code explicitly. With a closer look we notice that the sub-expression `d*D[i][k]+e*E[i][k]` is also invariant, although, this time, with respect to the outer (test functions) loop `j`. In [Luporini et al. 2014], we have showed that available compilers limit the search for code motion opportunities to the innermost loop of a given loop nest. Moreover, the lack of cost models to ascertain both the optimal place where to hoist an expression and whether or not vectorizing it at the price of extra temporary memory is a fundamental limiting factor. The *generalized loop-invariant code motion* technique that we implemented in COFFEE, leading to the code in Figure 2(b), addressed these limitations, which are critical in the case of non-trivial assembly expressions, resulting in run-time improvements up to $3\times$.

In this paper, we target particularly complex variational forms for which a more aggressive transformation is required. We generalize the problem of finding and hoisting invariant sub-expressions by formulating the following critical question: *how an assembly expression should be rewritten so as to minimize both floating point operations and data movement*? To minimize floating point operations, in terms of instructions executed, it is important to reduce the absolute number of arithmetic operations to

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

(a) Original (simplified) code

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

(b) The code after that generalized loop-invariant code motion has been applied

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

(c) Once terms are factorized, further code hoisting opportunities arise. Here, `T3` stores the pre-computation of an invariant sub-expression exposed by factorization

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

(d) Hoisting and scalar-expansion of integration-dependent terms allow to achieve full SIMD vectorization of the assembly code

Fig. 2: Applying a sequence of transformations to an excerpt from a Burgers' local assembly kernel, suitably simplified to improve readability.

evaluate the element matrix as well as to achieve full vectorization of the assembly loop nest.

We start considering the case of assembly expressions that, after generalized loop-invariant code motion has been applied, still "hide" opportunities for code hoisting. By examining again the code in Figure 2(b), we notice that the basis function array `A` iterating along the `[i,j]` loops appears twice in the expression. By expanding the products in which `A` is accessed and by applying sum commutativity, terms can be factorized. This has two effects: firstly, it reduces the number of arithmetic operations performed; secondly, and most importantly, it exposes a new sub-expression `A[i][k]/c+T2[k]*f` invariant with respect to loop `j`. Consequently, hoisting can be performed, resulting in the code in Figure 2(c). In general, exposing factorization opportunities requires traversing the whole expression tree, and then expanding and moving terms. It also needs heuristics to select a factorization strategy: there may be different opportunities of reorganizing sub-expressions, and, in our case, the best is the one that maximizes the invariant code eventually disclosed. We will discuss this aspect formally in Section 4.

As a second observation, we note that integration-dependent expressions are inherently executed as scalar code. For example, the `f0*a` and `f1*b` products in Figure 2(c) depend on the loop along quadrature points; these operations are performed in a non-vectorized way at every `i` iteration. This is not a big issue in our running example, in

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

(a) Original (simplified) code

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

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

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

(c) The code after expansion of `g`. Note the need to introduce a new temporary array

Fig. 3: Expansion of terms to improve register pressure in a local assembly kernel

which the scalar computation represents a small fraction of the total, but it becomes a concrete problem in complicated forms, like those at the heart of hyperelasticity (which we evaluate in Section 5). In such forms, the amount of computation independent of both test and trial functions loops is so large that it has a significant impact on the run-time, despite being executed only $O(I)$ times (with $I$ number of quadrature points). We have therefore implemented an algorithm to move and scalar-expand integration-dependent expressions, which leads to codes as in Figure 2(d).

The third motivation for expression rewriting concerns the register pressure in the innermost loop. Once the code has been optimized for arithmetic intensity, it is important to think about how the transformations impacted register allocation. Assume the local assembly kernel is executed on a state-of-the-art CPU architecture having 16 logical registers, e.g. an Intel Haswell. Each value appearing in the expression is loaded and kept in a register as long as possible. In Figure 2(d), for instance, the scalar value `g` is loaded once, whereas the term `det*W[i]` is precomputed and loaded in a register at every `i` iteration. This implies that at every iteration of the `[j,k]` loop nest, 12% of the available registers are spent just to store values independent of test and trial functions loops. In more complicated expressions, the percentage of registers destined to store such constant terms can be even higher. Registers are, however, a precious resource, especially when evaluating compute-intensive expressions. The smaller is the number of available free registers, the worse is the instruction-level parallelism achieved: for example, a shortage of registers can increase the pressure on the L1 cache (i.e. it can worsen data locality), or it may prevent the effective application of standard transformations, e.g. loop unrolling. We aim at relieving this problem by suitably expanding terms and introducing, where necessary, additional temporary values. We illustrate this in the following example.

Consider a variant of the Burgers local assembly kernel, shown in Figure 3(a). This is again a representative, simplified example. We can easily distribute `det*W[i]` over the three operands on the left-hand side of the multiplication, and then absorb it in the pre-computation of the invariant sub-expression stored in `T1`, resulting in code as in Figure 3(b). Freeing the register destined to the constant `g` is less straightforward: we cannot absorb it in `T1` as we did with `det*W[i]` since `T1` is also accessed in the `T1[j]*A[i][k]` sub-expression. The solution is to add another temporary as in Figure 3(c). Generalizing, this is a problem of data dependencies: to solve it, we use a dependency graph in which we add a direct edge from identifier `A` to identifier `B` to

denote that the evaluation of `B` depends on `A`. The dependency graph is initially empty, and is updated every time a new temporary is created by either loop-invariant code motion or expansion of terms. The dependency graph is then queried to understand when expansion can be performed without resorting to new temporary values. This aspect is formalized in the next section.

### 3.2. Rewrite Rules

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

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

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

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

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

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

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

Rewrite rules for expression rewriting are provided in Figure 4; obvious rules are omitted for brevity. The Expression Rewriter applies the rules while performing a

| *Rule* | *Precondition* |
|---|---|
| $[a_i \cdot b_j]_{(\sigma,G)} \rightarrow [a_i \cdot b_j]_{(\sigma,G)}$ | |
| $[(a_i + b_j) \cdot \alpha]_{(\sigma,G)} \rightarrow [(a_i \cdot \alpha + b_j \cdot \alpha)]_{(\sigma,G)}$ | |
| $[a_i \cdot b_j + a_i \cdot c_j]_{(\sigma,G)} \rightarrow [(a_i \cdot (b_j + c_j)]_{(\sigma,G)}$ | |
| $[a_i + b_i]_{(\sigma,G)} \rightarrow [t_i]_{(\sigma',G')}$ | $t_i = \sigma_{i_0}[t'_i/_{a_i + b_i}], G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, b_i)\})$ |
| $[(a_i \cdot b_j) \cdot \alpha]_{(\sigma,G)} \rightarrow [t_i \cdot b_j]_{(\sigma',G')}$ | $\sharp(b_j) > \sharp(a_i), t_i = \sigma_{i_0}[^{\sigma[\perp/_{a_i}]}/_{a_i \cdot \alpha}], a_i \notin in(G),$ $G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, \alpha)\})$ |
| $[(a_i \cdot b_j) \cdot \alpha]_{(\sigma,G)} \rightarrow [t_i \cdot b_j]_{(\sigma',G')}$ | $\sharp(b_j) > \sharp(a_i), t_i = \sigma_{i_0}[t'_i/_{a_i \cdot \alpha}], a_i \in in(G),$ $G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, \alpha)\})$ |

Fig. 4: Rewrite rules driving the rewriting process of an assembly expression.

depth-first traversal of the assembly expression tree. Given an arithmetic operation between two sub-expressions (i.e. a node in the expression tree), we first need to find an applicable rule. There cannot be ambiguities: only one rule can be matched. If the preconditions of the rule are satisfied, the corresponding transformation is performed; otherwise, no rewriting is performed, and the traversal proceeds. As examples, it is possible to instantiate the rules in the codes shown in Figures 2(a) and 3(a); eventually, the optimized codes in Figures 2(c) and 3(c) are obtained, respectively.

To what extent should rewrite rules be applied is a question that cannot be answered in general. In some problems, a full rewrite of the expression may be the best option; in other cases, on the other hand, an aggressive expansion of terms, for example, may lead to high register pressure in the loops computing invariant terms, worsening the performance. In Section 4.3 we explain how to leverage the Code Specializer to select a suitable rewriting strategy for the problem at hand.

### 3.3. Avoiding Iteration on Zero-valued Blocks by Symbolic Execution

We aim at skipping arithmetic operations over blocks of zero-valued entries in basis functions (and their derivatives) arrays. Zero-valued columns arise, for example, when taking derivatives on a reference element and when employing mixed elements. In [Olgaard and Wells 2010], a technique to avoid operations on zero-valued columns based on the use of indirection arrays was described (e.g. `A[B[i]]`, where `A` is a tabulated basis function and `B` a map from loop iterations to non-zero columns in `A`) and implemented in the context of FEniCS. We will evaluate our approach and compare it to this pioneering work in Section 5. Essentially, our strategy avoids indirection arrays in the generated code, which otherwise would break the optimizations applicable at the Code Specializer level, including SIMD vectorization.

Consider Figure 5(a), an enriched version of the Burgers excerpt in Figure 2(b). The code is instantiated for the specific case of polynomial order 1, Lagrange elements on a 2D mesh. The array `D` represents a tabulated derivative of a basis function at the various quadrature points. There are four zero-valued columns. Any multiplications or additions along these columns could (should) be skipped to avoid irrelevant floating point operations. The solution adopted in [Olgaard and Wells 2010] is not to generate the zero-valued columns (i.e. to generate a dense $6 \times 2$ array), to reduce the size of the

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

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

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

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

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

(b) The code after symbolic execution took place

Fig. 5: Simplified excerpt of local assembly code from a Burgers form using vector-valued basis functions, before and after symbolic execution is performed to rewrite the iteration space

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

Our approach is based on using domain knowledge and symbolic execution. We discern the origin of zero-valued columns: for example, those due to taking derivatives on the reference element from those inherent to using mixed (vector) elements. In the running Burgers example, the use of vector function spaces require the generation of a zero-block (columns 0, 1, 2 in the array `D`) to correctly evaluate the local element matrix while iterating along the space of test and trial functions. The two key observations are that 1) the number of zero-valued columns caused by using vector function spaces is, often, much larger then that due to derivatives, and 2) such columns are contiguous in memory. Based on this observation, we aim at avoiding iteration only along the block of zero-valued columns induced by mixed (vector) elements.

We achieve our goal by means of symbolic execution. The Expression Rewriter expects some indication about the location of the zero-valued columns induced by mixed (vector) function spaces, for each tabulated basis function. This indication comes either in the form of code annotation, if the input to COFFEE were provided as pure C (as shown in Figure 5(a)), or by suitably decorating basis functions nodes, if the input were an abstract syntax tree. Then, the rewrite rules are applied and each statement is executed symbolically. For example, consider the assignment `T2[r] = d*D[i][k]+e*E[i][k]` in Figure 5(a). Array `D` has non-zero-valued columns in the range $NZ_D = [3, 5]$; we also assume array `E` has non-zero-valued columns in the range $NZ_E = [0, 2]$. Multiplications by scalar values do not affect the propagation of non-

zero-valued columns. On the other hand, when symbolically executing the sum of the two operands `d*D[i][k]` and `e*E[i][k]`, we track that the target identifier `T2` will have non-zero-valued columns in the range $NZ_E \| NZ_D = [0, 5]$. Eventually, exploiting the $NZ$ information computed and associated with each identifier, we split the original assembly expression into multiple sets of sub-expressions, each set characterized by the same range of non-zero-valued columns. In our example, assuming that $NZ_{T1} = [3, 5]$ and $NZ_A = [3, 5]$, there are two of such sets, which leads to the generation of two distinct iteration spaces (one for each set), as in Figure 5(b).

## 4. CODE SPECIALIZATION

Code specialization provides a range of code transformations tailored to optimizing for instruction-level parallelism and register locality. As summarized in Section 2.2 and described in [Luporini et al. 2014], padding and data alignment, expression splitting, and outer-product vectorization, which is a domain-aware implementation of vector-register tiling, are examples of optimizations that the Code Specializer is capable of applying. In this paper, we revisit and enrich this set of transformations as explained in the following sections.

### 4.1. Padding and Data Alignment Revisited

Padding and data alignment as described in [Luporini et al. 2014] must be refined for the case in which computation over zero-valued columns is avoided. We recall effective SIMD (auto-)vectorization can be achieved only if the innermost loop size is a multiple of the vector register length, in which case the compiler needs not to introduce a remainder scalar loop. In the case of an AVX instruction set, for example, we want to round the size of the innermost loops to the closest multiple of 4 (AVX registers can fit up to four double-precision floats). This can be done provided that arrays are padded, if necessary. Moreover, loads and stores instructions are efficient only if their addresses are suitably aligned to cache boundaries; this is obtainable by enforcing the base address of each padded array to be a multiple of the vector length.

Consider again the code in Figure 5(b). The arrays in the loop nest `[j1,k1]` can be padded and the right bound of loop `k1` can be safely increased to 8: eventually, values computed in the region `M[0:3][6:8]` will be discarded. Then, by explicitly aligning arrays and using suitable pragmas (e.g. `#pragma simd` for the Intel compiler), effective SIMD auto-vectorization can be obtained for this loop nest.

There are some complications in the case of loops `[j0,k0]`. Here, increasing the loop bound to 4 is still safe, assuming that both `T1` and `A` are padded with zero-valued entries, but it has no effect: the starting addresses of the load instructions would be `T1[3]` and `A[i][3]`, which are not aligned. One solution is to start iterating from the closest index that would ensure data alignment: in this specific case, $k0 = 0$. However, this would imply losing (partially in general, totally for this loop nest) the effect of the zero-avoidance transformation. Another possibility is to attain to non-aligned accesses. COFFEE can generate code for both situations, so we leave the autotuner, described in Section 4.3, in charge of determining the optimal transformation.

Note that in some circumstances the previous solution cannot be applied, since extra iterations might end up accessing non-zero entries in the local element matrix `M`. In such a situation, there is no possibility of recovering data alignment.

### 4.2. Exposing Linear Alegbra Operations

In [Luporini et al. 2014], we introduced the idea of transforming the element matrix evaluation into a sequence of calls to highly-optimized dense matrix-matrix multiply routines (henceforth DGEMM), for instance MKL or ATLAS BLAS. We compared COFFEE's optimizations with hand-written MKL-based kernels, showing how the small

sizes of the involved arrays impair DGEMM routines, which are usually tuned for large arrays. The study was conducted only in the case of a specific Helmholtz form. There are scenarios, however, in which tabulated basis functions sizes grow up to a point for which turning to BLAS may actually lead to better performance. For example, this can happen when relatively-high polynomial orders are used or if coefficients are present in the form, since they are both responsible for the size of basis functions tabulated at quadrature points. To explore these scenarios, we have developed an algorithm that reduces any assembly expression evaluation to a sequence of DGEMM calls.

We informally provide the main steps of the algorithm. By fully applying the rewrite rules in Figure 4, an assembly expression is reduced to a summation, over each quadrature point, of outer products along the test and trial functions. Each outer product is then isolated, i.e. the assembly expression is split into chunks, each chunk representing an outer product over test and trial functions. Statements in the bodies of the surrounding loops (e.g. coefficients evaluation at a quadrature point, temporaries introduced by expression rewriting) are vector-expanded and hoisted completely outside of the loop nest, similarly to what we have described in Section 3.1. This renders the loop nest perfect; that is, there is no intervening code among the various loops. The element matrix evaluation has now become a sequence of dense matrix-matrix multiplies (transposition aside)

$$A_{jk} = \sum_i x0_{i_j} \cdot y0_{i_k} + \sum_i x1_{i_j} \cdot y1_{i_k} + ...$$

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

### 4.3. Model-driven Dynamic Autotuning

We have demonstrated in [Luporini et al. 2014] that determining the sequence of transformations that maximizes the performance of a problem requires investigating a broad range of factors, including mathematical structure of the input form, polynomial order of employed function spaces, presence of pre-multiplying functions, and, of course, the characteristics of the underlying architecture.

In [Luporini et al. 2014], we proposed a simple cost model to drive the optimization process. In this paper, we have added a significant number of options to the set of possible optimizations, so the selection problem is now far more challenging. The sole Expression Rewriter, for instance, can generate a wide variety of implementations by just applying rewrite rules to different extents. We found difficult extending the cost model to effectively cover such a large search space.

We tackle the optimizations selection problem by compiler autotuning. Not only does it allow to determine the best combination of transformations, also it enables exploring parametric low-level optimizations, such as loop unroll, unroll-and-jam, and interchange, by trying different unroll factors and loop permutations. By leveraging the cost model defined in our previous study, domain knowledge, and a set of heuristics, we manage to keep the autotuner overhead at a minimum, whilst achieving significant speed ups over the purely cost-model-based implementations in all of the forms we have evaluated. In particular, our autotuner usually requires order of seconds to determine the fastest kernel implementation, a negligible overhead when it comes to

iterate over real-life unstructured meshes, which can contain up to trillions of elements (e.g. [Rossinelli et al. 2013]).

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

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

— Loop permutations that are likely to worsen the performance are excluded from the search space. According to the cost model, and for the same reasons explained in [Luporini et al. 2014], we enable only variants in which the loop over quadrature points is either the outermost or the innermost. This is due to the fact that versions of the code in which such loop lies between the test and trial functions loops are typically lower performing.
— The unroll factors must divide the loop bounds evenly to avoid the introduction of reminder (scalar) loops.
— The innermost loop is never explicitly unrolled. This is because we expect auto-vectorization along this loop, so memory accesses should be kept unit-stride.

The autotuner is also domain-aware: the following heuristics, which capture properties of the computational domain, are exploited

— The lengths of test and trial functions loops are identical in some cases, for example when they originate from the same function space. In such cases, since for the employed storage layout the memory access pattern is symmetrical along these two loops, we prune their interchange from the search space.
— The larger is the polynomial order of the method, the larger is the assembly loop nest. In these cases, we impose a bound on the loop nest's overall unroll factor (which we found empirically) to avoid uselessly testing too many unroll factors.
— On the other hand, if the polynomial order is low, i.e. when the loop nest is small, we prune variants that we know will be low-performing, e.g. those resorting to BLAS.
— We specifically select two levels of expression rewriting. In the "base" level, only generalized loop-invariant code motion, as described in [Luporini et al. 2014], is applied. This means that only a subset of the rewrite rules exposed in 4 will be considered. In the "aggressive" level, all of the rewrite rules are applied. Many other trade-offs, which we do not explore, would be feasibile, however.
— For the expression splitting optimization described in [Luporini et al. 2014] and summarized in Section 2.2, we test only three split factors, namely 1, 2, 4. Also, if the input problem uses mixed function spaces, the iteration space is already split at expression rewriting time to avoid computation over zero-valued columns; in these cases, we do not further apply expression splitting.

— Based on the cost model, the padding and data alignment optimization is always applied.

All of the previous points contribute to minimize the overhead of the autotuning process. We will discuss the actual cost of autotuning in Section 5.3.

## 5. PERFORMANCE EVALUATION

### 5.1. Experimental Setup

Experiments were run on a single core of an Intel architecture, a Sandy Bridge I7-2600 CPU, running at 3.4GHz, 32KB L1 cache and 256KB L2 cache. The Intel `icc 14.2` compiler was used. The compilation flags used were `-O3, -xHost, -xAVX, -ip`.

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

We evaluate the *speed ups* achieved by three sets of optimizations over the original code; that is, the code generated by the FEniCS Form Compiler when no optimizations are applied. In particular, we analyze the impact of the FEniCS Form Compiler's built-in optimizations (henceforth `ffc`), the impact of COFFEE's transformations as presented in [Luporini et al. 2014] (referred to as `fix`, in the following), and the effect of Expression Rewriting and Code Specialization as described in this work (henceforth `auto`, to denote the use of autotuning as described in Section 4.3). The `auto` values do not include the autotuner cost, which is commented aside in Section 5.3.

The values that we report include the cost of local assembly as well as the cost of matrix insertion. However, the unstructured mesh has been made small enough to fit the L3 cache, so as to minimize the "noise" due to any operations that are not part of the element matrix evaluation itself. However, it has been reiterated over and over (e.g. [Olgaard and Wells 2010]) that as the complexity of a form increseas, the cost of local assembly becomes dominant. All codes were executed in the context of the Firedrake framework.

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

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

— The polynomial order of basis functions, $q \in \{1, 2, 3, 4\}$
— The polynomial order of coefficient (or "pre-multiplying") functions, $p \in \{1, 2, 3, 4\}$
— The number of coefficient functions $nf \in \{0, 1, 2, 3\}$

On the other hand, other aspects are fixed

— The space of both basis and coefficient functions is Lagrange
— The mesh is three-dimensional, made of tetrahedrons, for a total of 4374 cells

Figures 6, 7, 8, and 9, which will be deeply commented in the next section, must be read as "plots, or grids, of plots". Each grid (figure) has two logical axes: $p$ varies along the horizontal axis, while $q$ varies along the vertical axis. The top-left plot in a grid shows speed ups for $[q = 1, p = 1]$; the plot on its right does the same for $[q = 1, p = 2]$, and so on. The diagonal of the grid shows plots for which basis and coefficient functions have same polynomial order, that is $q = p$. Therefore, a grid can be read in many different ways, which allows us to make structured considerations on the effect of the various optimizations.

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

### 5.2. Performance of Forms

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

We first comment on results of general applicability. By looking at the various figures, we note there is a trend in COFFEE's optimizations to become more and more effective as $q$, $p$, and $nf$ increase. This is because most of the transformations applied aim at optimizing for arithmetic intensity and SIMD vectorization, which obviously have a strong impact when arrays and iteration spaces are large. The corner cases of this phenomenon are indeed $[q = 1, p = 1]$ and $[q = 4, p = 4]$. We also observe how `auto`, in almost all scenarios, outperforms all of the other variants. In particular, it is not a surprise that `auto` is faster than `fix`, since `fix` is one of the autotuner's tested variants, as explained in Section 4.3. This proves the quality of the work presented in this paper, which shows significant advances over [Luporini et al. 2014]. The reasons for which `auto` exceeds both original code and `ffc` are discussed for each specific problem next. Also, details on the "optimal" code variant determined by autotuning are given in Section 5.3.

*Helmholtz.* The results for the Helmholtz problem are provided in Figure 6. We observe that `ffc` slows the code down, especially for $q \geq 3$. This is a consequence of using indirection arrays in the generated code that, as explained in Section 3.3, prevent, among the other compiler's optimizations, SIMD auto-vectorization. The `auto` version results in minimal performance improvements over `fix` when $nf = 0$, unless $q = 4$. This is due to the fact that if the loop over quadrature points is relatively small, then close-to-peak performance is obtainable through basic expression rewriting and code specialization; in this circumstance, generalized loop-invariant code motion and padding plus data alignment. The trend changes dramatically as $nf$ and $q$ increase: a more ample spectrum of transformations must be considered to find the optimal local assembly implementation. We will provide details about the selected transformations in the next section.

*Elasticity.* Figure 7 illustrates results for the Elasticity problem. This form uses a vector-valued space for the basis functions, so here transformations avoiding computation over zero-valued columns are of key importance. The `ffc` set of optimizations leads to notable improvements over the original code at $q = 1$. The use of inderection arrays allows to phisically eliminate zero-valued columns at code generation time; as a consequence, different tabulated basis functions are merged into a single array. Therefore,

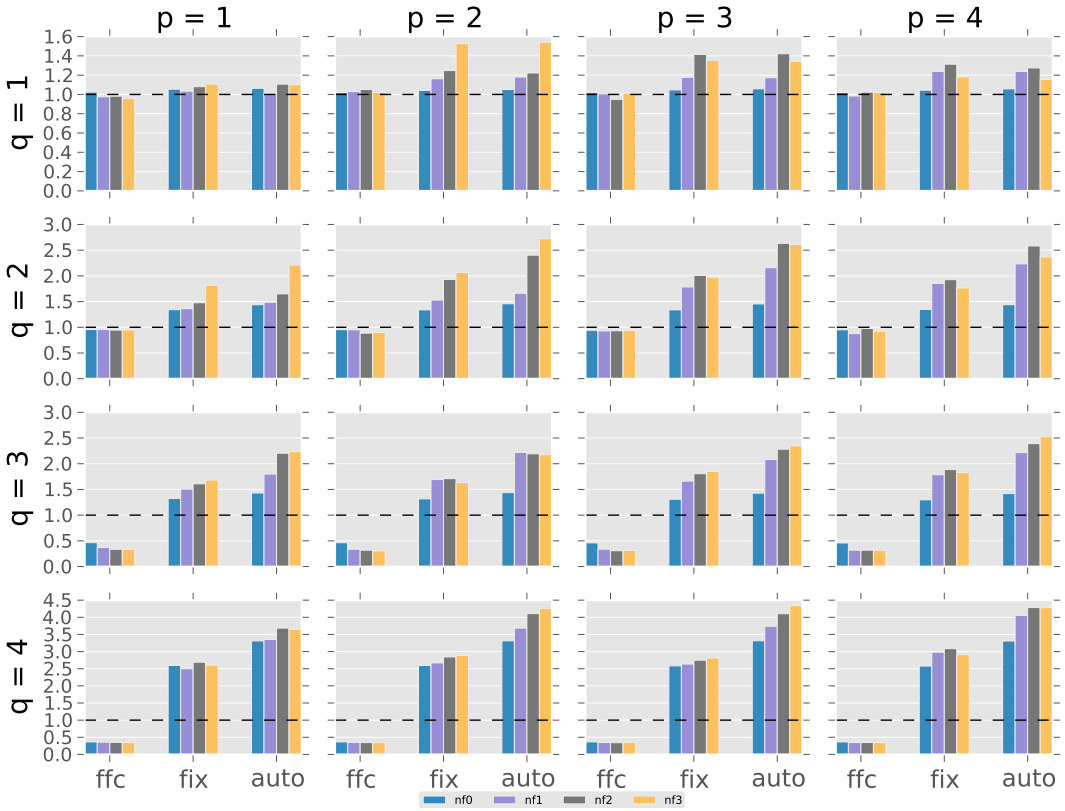


Fig. 6: Helmholtz results.

despite the execution being purely scalar because of indirection arrays, the reduction in arithmetic intensity and register pressure imply improvement in performance. Nevertheless, `auto` remains in general the best choice, with gains over `ffc` that are wider as $p$ and $nf$ increase.

For $q \geq 2$, in `ffc` the lack of SIMD vectorization counterbalances the decrease in the number of floating point operations, leading to speed ups over the original code that only occasionally exceed $1\times$. On the other hand, the successful application of the zero-avoidance optimization while preserving code specialization plays a key role for `auto`, resulting in much higher performance code especially at $q = 2$ and $q = 3$.

It is worth noting that speed ups of `auto` over `fix` decrease at $q = 4$, particularly for low values of $p$. As we will discuss in Section 5.3, this is because at $q = 4$ the vector-register tiling transformation (in combination with loop unroll-and-jam) leads to the highest performance. In principle, vector-register tiling can be used in combination to the zero-avoidance technique; however, due to mere technical limitations, this is currently not supported in COFFEE. Once solved, we expect much higher speed ups in the $q = 4$ regime as well.

*Poisson.* In Figure 8 we report speed ups of `ffc`, `fix`, and `auto` over the original code for the Poisson form. We note that, as a general trend, `ffc` exhibits drops in performance as $nf$ increases, notably when $nf = 3$, for any values of $q$ and $p$. This is a consequence of the inherent complexity of the generated code. The way `ffc` performs loop-

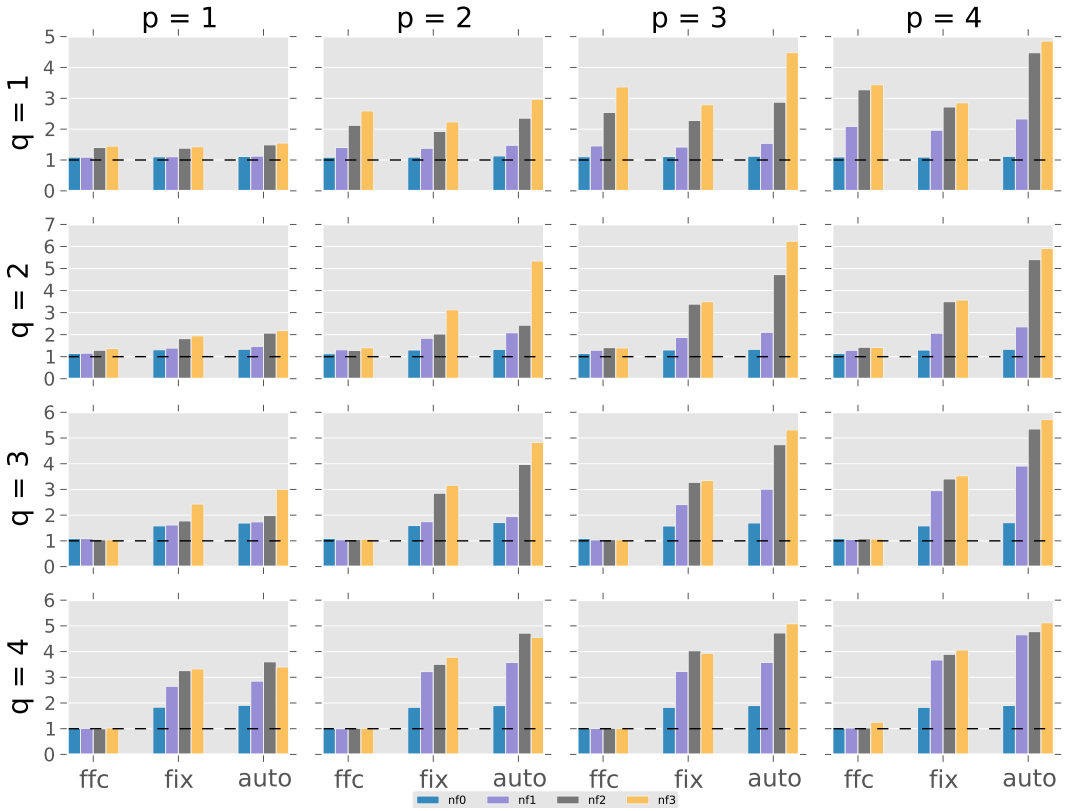


Fig. 7: Elasticity results.

invariant code motion leads to the pre-computation of integration-dependent terms at the level of the integration loop, which are characterized by higher arithmetic intensity and redundant computation as $nf$ increases. Moreover, the absence of vectorization is another limiting factor.

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

*Hyperelasticity*. Speed ups for the hyperelasticity form are shown in Figure 9. Experiments for $nf \geq 2$ could not be executed because of FEniCS-Form-Compiler's technical limitations.

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

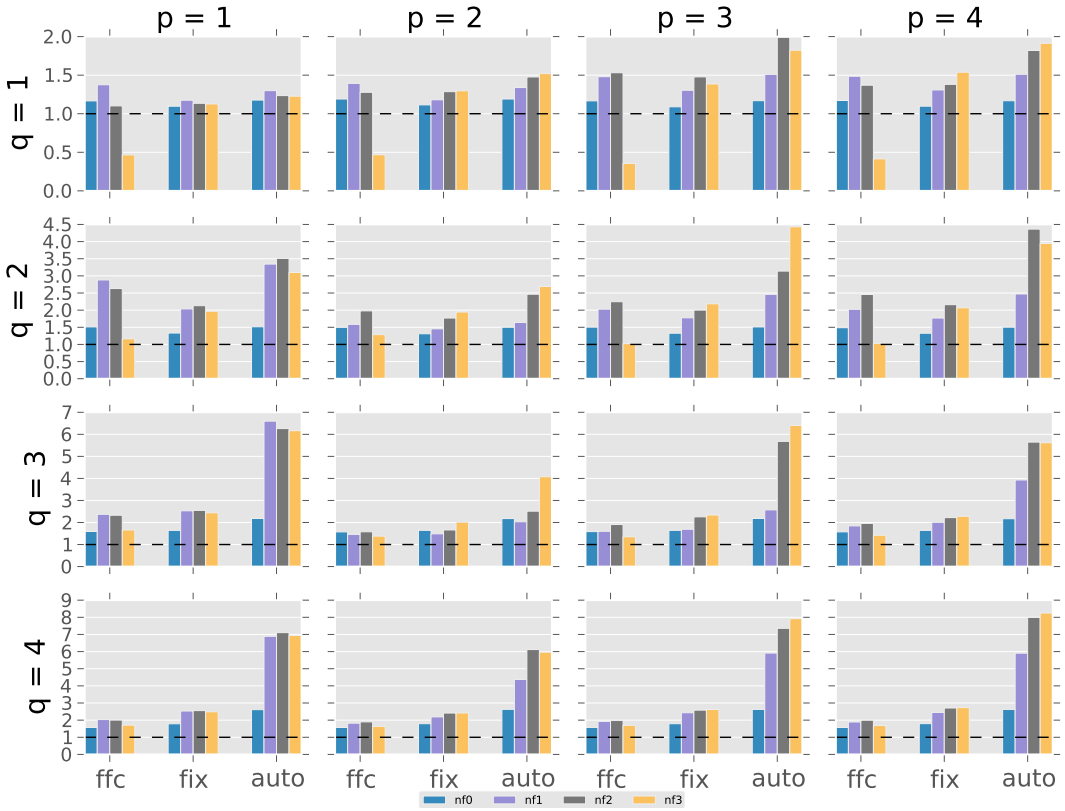


Fig. 8: Poisson results.

register usage). We expect to obtain considerably faster code once such optimizations will be incorporated.

In the regime $q \geq 2$ and $nf = 1$, peformance improvements are less pronounced moving from $p = 1$ to $p = 2$, although still significant; in particular, we notice a drop at $p = 2$, followed by a raise up to $p = 4$. It is worth observing that this effect is common to all sets of optimizations. The hypothesis is that this is due to the way coefficient functions are evaluated at quadrature points (identical in all configurations), which cannot be easily vectorized unless a change in storage layout and loops order is implemented in the code (abstract syntax tree) generator on top of COFFEE.

### 5.3. Details on the Autotuning Process

*Autotuning Overhead.* We first comment on the overhead of the autotuning process. In the context of Firedrake, the framework in which COFFEE is integrated, the autotuner is executed at run-time, once the local assembly kernels are provided by the FEniCS Form Compiler. Autotuning, therefore, introduces an overhead in the application execution time. Such overhead originates from four operations: 1) creation of the various code variants; 2) generation of a C file containing such variants (as simple function calls, plus the "main" function that invokes the variants, in sequence); 3) compilation of the autotuning file; 4) execution. Of these four points, we note that: the cost of 4) is relatively small, because each variant's execution time is bound by an empirically-found value (e.g. some milliseconds). The cost of all four points is constrained by our

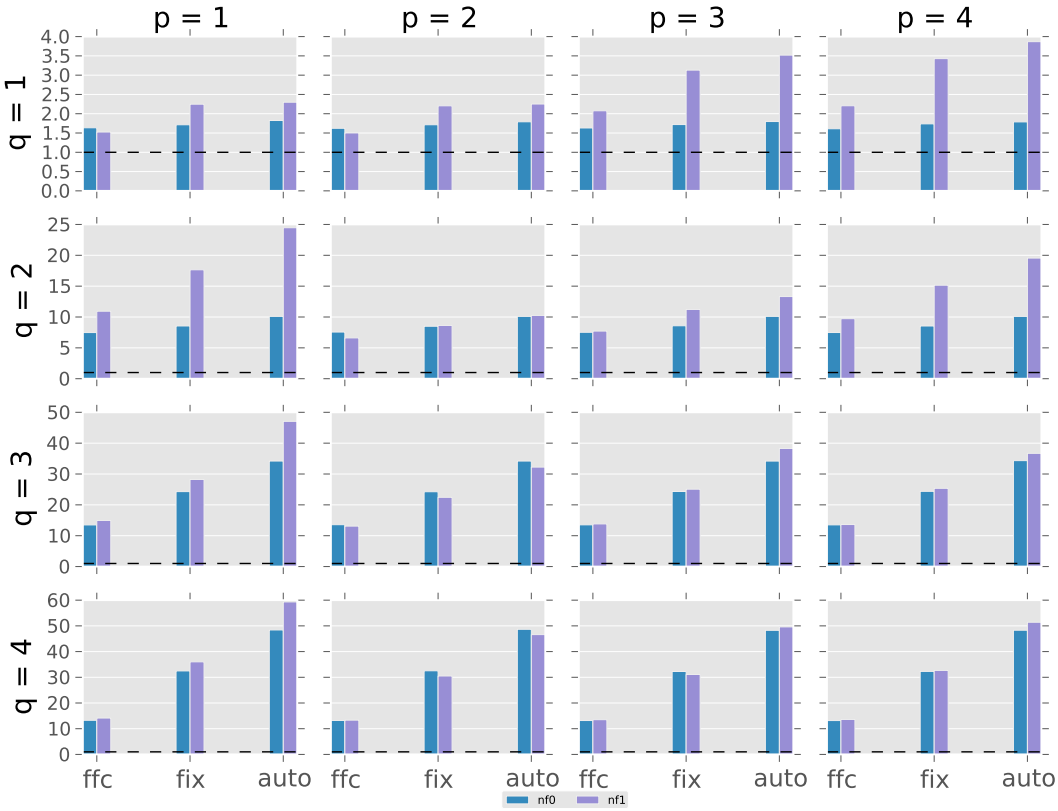


Fig. 9: Hyperelasticity results.

heuristics to prune the search space, as described in Section 4.3. Moreover, for a given form and a given discretization, the autotuner needs to be executed only once, since its output is saved and reused for later assemblies. This implies that if the assembly occurs in a time stepping loop, or the same form is executed on a different mesh, or known quantities of the input problem are changed, then the assembly cost is rapidly amortized. Premised that, the most important thing remains that when working with real unstructured meshes - which are likely to be composed of millions of elements, leading to long-lasting assembly phases - the autotuner overhead practically becomes completely negligible. In our experience, and in particular in the four examined problems, the autotuning process lasted less than a minute in the majority of cases. Most often, it took less than thirty seconds. Rarely it needed more than a minute, as for instance in the case of hyperelasticity; however, despite the inherent complexity of this form, we measured a peak of only 4 minutes for the extreme case $[q = 4, p = 4, nf = 1]$, while 1.30 minutes were needed in the case $[q = 1, p = 1, nf = 1]$. This analysis certifies that the autotuner overhead is definitely sustainable in real-world applications.

*Selected Optimizations.* In this paper and in [Luporini et al. 2014], we have repeatedly claimed that different forms (and different discretizations) require distinct sets of transformations to reach close-to-peak performance. To demonstrate this, we now report details about the output of the autotuning process. We show that for the four examined forms - more specifically, for the 224 problem instances $[form, q, p, nf]$ illus-

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

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

trated in the previous figures - a plethora of optimization strategies have been selected by the autotuner. We also complement and strengthen our claim by showing the autotuner's output for two additional forms whose performance results, for brevity, were not shown in Section 5.2: a Mass and a Mixed Poisson problems.

Table I shows the number a given transformation has been selected by the autotuner. To not hinder readability, the output has been grouped by form, rather than showing the selected optimization strategy for each $[form, q, p, nf]$. Values in the `rewrite strategy` column can either be `base` or `aggressive`, as explained in Section 4.3: in the former case, only generalized loop-invariant code motion in applied; in the latter case, the rewrite rules are recursively applied as extensive as possible. If the value is `aggressive` (`base`) it means that in the majority of cases the `aggressive` (`base`) strategy prevailed on the `base` (`aggressive`) one. Precomputation of integration-dependent terms was explained in Section 3.1. The `split` column refers to the expression splitting transformation ([Luporini et al. 2014], and summary in Section 2.2). The `unroll` column indicates the application of explicit unrolling. Other columns are of obvious meaning. Values in the various cells illustrate the number of problem instances (out of the total reported in column `number of variants`) in which an optimization was activated. Obviously, more transformations are typically used in combination in a same problem.

## 6. CONCLUSIONS

We started discussing how complex is the problem of optimizing local assembly operations in the context of automated code generation, in which an infinite range of possibly extremely complicated variational forms need to be supported. Available compilers and third-party libraries cannot exploit the structure inherent assembly expressions, and most of the time fail at producing efficient code. We have demonstrated that when optimizing local assembly kernels three points must be addressed: minimization of floating point operations, instruction-level parallelism (especially SIMD vectorization) and data (register) locality. Our approach is based on a two-stage optimization procedure: expression rewriting and code specialization. The first stage produces code that suits optimizations in the second stage. Our motivation for this structured strategy was that we felt state-of-the-art optimization systems were not adequate enough for moderately and remarkably complex variational forms.

We have implemented the Expression Rewriter and the Code Specializer in COFFEE, a platform-independent optimizing compiler tailored to local assembly operations, integrated with the popular Firedrake framework. This, along with the extensive and successful performance evaluation provided, prove the maturity of our work.

## REFERENCES

2014. UFL forms. https://github.com/firedrakeproject/firedrake-bench/blob/experiments/forms/firedrake_forms.py. (2014).

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

Krzysztof Bana, Przemyslaw Plaszewski, and Pawel Maciol. 2014. Numerical Integration on GPUs for Higher Order Finite Elements. *Comput. Math. Appl.* 67, 6 (April 2014), 1319–1344. DOI:http://dx.doi.org/10.1016/j.camwa.2014.01.021

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 101–113. DOI:http://dx.doi.org/10.1145/1375581.1375595

Firedrake contributors. 2014. The Firedrake Project. http://www.firedrakeproject.org. (2014).

Robert C. Kirby, Matthew Knepley, Anders Logg, and L. Ridgway Scott. 2005. Optimizing the Evaluation of Finite Element Matrices. *SIAM J. Sci. Comput.* 27, 3 (Oct. 2005), 741–758. DOI:http://dx.doi.org/10.1137/040607824

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

A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. 2009. Nodal Discontinuous Galerkin Methods on Graphics Processors. *J. Comput. Phys.* 228, 21 (Nov. 2009), 7863–7882. DOI:http://dx.doi.org/10.1016/j.jcp.2009.06.041

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

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

Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. 2014. COFFEE: an Optimizing Compiler for Finite Element Local Assembly. (2014).

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

Diego Rossinelli, Babak Hejazialhosseini, Panagiotis Hadjidoukas, Costas Bekas, Alessandro Curioni, Adam Bertsch, Scott Futral, Steffen J. Schmidt, Nikolaus A. Adams, and Petros Koumoutsakos. 2013. 11 PFLOP/s Simulations of Cloud Cavitation Collapse. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 3, 13 pages. DOI:http://dx.doi.org/10.1145/2503210.2504565

Francis P. Russell and Paul H. J. Kelly. 2013. Optimized Code Generation for Finite Element Local Assembly Using Symbolic Manipulation. *ACM Trans. Math. Software* 39, 4 (2013).

Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul D. Hovland. 2010. Speeding Up Nek5000 with Autotuning and Specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, New York, NY, USA, 253–262. DOI:http://dx.doi.org/10.1145/1810085.1810120