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

Abstract goes here

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), 15 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 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

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

spaces, the resulting assembly kernels become more and more expensive, covering a significant fraction of overall computation run-time.

Achieving 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. Also, traditional vendor compilers, such as *GNU's* and *Intel's*, fail at exploiting the structure inherent assembly expressions. This has led to development of a number of higher-level approaches to optimize 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 the problem. In [Luporini et al. 2014], we have made one step forward by showing that different problems, on different platforms, require distinct set of transformations if close-to-peak performance needs to be obtained, 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 GPUs, for instance in [Knepley and Terrel 2013].

Our research has resulted in the development of a compiler, COFFEE[1], fully integrated with the Firedrake framework. While clearly separating the mathematical domain, which remains captured in the higher-level form compiler, from the optimization process, COFFEE also aims to be platform-agnostic. The code transformations occur on an intermediate representation of the assembly code, which is ultimately translated into platform-specific code. Domain knowledge is exploited in two ways: for simplifying the implementation of a broad range of code transformations, and, obviously, 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 the work presented in [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 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 obtain, for instance, 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 sub-steps of code specialization; on the other hand, generalized loop-invariant code motion is a step of the expression rewriting process. More importantly, we complement and generalize our previous work with the following contributions.

Expression rewriting is based on a formal set of rewrite rules. Our first contribution consists of a framework that aggressively exploits associativity, distributivity, and commutativity of arithmetic operators to expose "hidden" loop-invariant sub-expressions.

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

Secondly, we show how to make use of domain knowledge to avoid computation over zero-valued regions 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 by using FEniCS' built-in optimizations, 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 show the benefit of vector-expansion to achieve SIMD vectorization of otherwise scalar code. This is particularly useful in complex forms, like those based on hyperelasticity. Secondly, we answer 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 multiplies. 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 important cases, especially in forms characterized by pre-multiplying functions and relatively high-order function spaces, a BLAS-based execution strategy can be a successful. Finally, we introduce a model-driven, dynamic autotuner that automatically and transparently compose the set of code transformations that are likely to maximize the performance of a given problem. The main challenge with the autotuner is to maintain, for any possible problem, the search space reasonably small, although comprising the most effective code variants, so that the overhead, which impacts the run-time, is negligible.

Expression rewriting and code specialization have been implemented in COFFEE and are fully operating. Therefore, to testify the goodness of our approach, we provide an extensive and unprecedented performance evaluation in a number of forms of increasing complexity, including problems based on hyperelasticity operators. 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 four sets of code variants, for each problem instance: 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 detailed.

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 real example, the evaluation of the local assembly (or, equivalently, 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 Ω 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 $F_K : K_0 \rightarrow K$ 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 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 Ω, 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* of the 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-

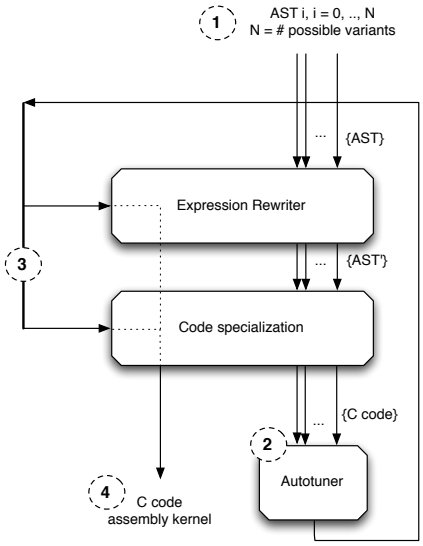

Fig. 1: Outline of COFFEE

level parallelism, and 3) data locality. In this paper, we tackle these three points building on our previous work ([Luporini et al. 2014]).

COFFEE is a mature, platform-independent compiler capable of optimizing local assembly code. Its high-level structure can be outlined as in Figure 1. The input is either suitably-annoted C code, explicitly provided by the user, or a decorated abstract syntax tree (henceforth AST) representation of an assembly kernel, automatically generated by the higher level in the stack. We have integrated COFFEE with the Firedrake framework; here, the input to COFFEE is originated by the FEniCS Form Compiler 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 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. 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 in the expression rewriting stage do 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 [Luporini et al. 2014] (as well as to FEniCS' built-in optimizations).

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 subset of optimizations are needed to achieve close-to-peak performance in different problems. 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 briefly described in Section 3, we work around this limitation, while also achieving vectorization of invariant code.
— *Padding and data alignment*. The small size of the loop nest 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 logically applied in COFFEE; also, the contributions of this work are highlighted. In the next two sections, we describe the new functionalities of, respectively, the Expression Rewriter and the Code Specializer.

3. EXPRESSION REWRITING

As summarized in 2.2, loop-invariant code motion is the key to reduce the computational intensity of an assembly expression. The Expression Rewriter (henceforth ER) that we have designed and implemented in COFFEE enhances this technique by making two steps forward, which allow more redundant computation to be avoided.

Firstly, exploiting arithmetic operations properties like associativity, distributivity, and commutativity, it manipulates the original expression to expose more opportunities to the code hoister. There are many possibilities of rewriting an expression, and the search space can quickly become too big. Therefore, one problem we solve is finding a sufficiently simple yet systematic way of maximizing the amount of loop-invariant operations in an expression. In Section 3.1, we formalize the set of rewrite rules that COFFEE follows to transform an expression.

Secondly, the ER re-structures the loop nest so as to eliminate arithmetic operations over array columns that are statically known to be zero-valued. Zero 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 it is that by using indirection arrays in the generated code, it breaks many of the optimizations that can be applied at the Code Specializer level, including SIMD vectorization. In Section 3.3, we show a novel approach to avoiding computation on zeros based on symbolic execution.

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

(a) Original code

```
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]));
    T2[r] = ((d*D[i][k])+(e*E[i][k]));
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += ((T1[j]*A[i][k]*g)+
                 ((A[i][k]/c)+(T2[k]*f))*A[i][j])*det*W[i];
    }
  }
}
```

(b) Factorized code

Fig. 2: Original and factorized code.

3.1. Objectives of the Expression Rewriter

Consider the simplified example of the element matrix computation in Figure 2(a), which is an excerpt from a real Burgers problem. In practice, depending on the problem, the expression tree could be much more complex, with multiple levels of nesting. The example is however representative for a large class of problems, so we will use it throughout the rest of the paper for illustrative purpose.

A first glimpse of the code suggests that the sub-expression `F[i][j]*d + G[i][j]*e` is invariant with respect to the innermost loop, so it should be hoisted at the level of the outer loop `j`. This is a standard compiler transformation, which is supported by any available compilers. With a closer look we notice that the sub-expression `a*C[i][k] + b*E[i][k]` is also invariant, although, in this case, with respect to the outer loop `j`. In [Luporini et al. 2014], we have showed that a *generalized* loop-invariant code motion transformation - that is, given a non-trivial expression, the capability of analyzing all of its sub-expressions with respect to the enclosing loops to determine what code is hoistable- is not supported by available compilers. 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. We have addressed these problems by implementing a generalized loop-invariant code motion transformation in COFFEE.

We now consider the case of forms that "hide" further opportunities for code hoisting. Such forms are by no means exotic: for example, the pattern described next is commonly found in elastic problems. By examining again the example in Figure 2(a), we notice that the basis function array `B`, iterating along the iteration space `[i,j]`, appears twice in the expression. If we expand the products in which `B` is accessed, we can apply product commutativity and then factorize the expression as in Figure 2(b). This has two effects: firstly, it reduces the total 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`. Therefore, code hoisting can be performed.

The second observation we make concerns the register pressure induced by the expression. Once loop-invariant terms are lifted, we can think about data locality and, in particular, register allocation. Assume the local assembly kernel is executed on a state-of-the-art 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. 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 `jk` loop nest, 12% of the available registers are spent just to store constant values. In more complicated expressions, the percentage of registers destined to store loop-invariant terms can be even higher. Registers are, however, a precious resource, especially when evaluating intensive expressions. The smaller is the number of free registers, the worse is the instruction-level parallelism achieved: for example, a

```
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];
    }
  }
}
```

Fig. 3: Expandable 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]);
    }
  }
}
```

(a) Expanded 1 code

```
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]);
    }
  }
}
```

(b) Expanded 2 code

Fig. 4: Expanded code.

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 like loop unrolling. The ER works around this problem by suitably expanding terms and introducing, where necessary, new temporary values.

Consider the variant of the running (transformed) example shown in Figure 3. Again, this is a representative example of what happens in real finite element forms. 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 `T1`, resulting in the code illustrated in Figure 4(a). Freeing the register destined to the constant `g` is more complicated: we cannot absorb it in the pre-computation of `T1` because the same array is accessed in the evaluation of `(T1[ j]*A[i][k])`. The solution is to add another temporary as in Figure 4(b). Generalizing, this is a problem of data dependencies; in order to solve it, we employ 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; every time a new temporary is created due to loop-invariant code motion or expansion of terms is performed, it is updated by suitably adding vertices and edges.

3.2. Rewrite Rules

In general, assembly expressions produced by automated code generation can be much more complex (more terms and operations involved) and nested. Our goal is to establish a portable, platform- and compiler-independent, and systematic way of reducing the strength of an expression. The technique should be simple; definitely it must be robust to be integrated in an optimizing domain-specific compiler capable of supporting real problems. Ideally, it should be naturally extendible to problems that will be supported in next releases of state-of-the-art frameworks like Firedrake and FEniCS: for instance, explicit support for outer-product finite elements will enable generation of kernels with much deeper loop nests, and the ER should transparently be able to deal with these structures as well.

To address these issues, we have based the implementation of the ER in COFFEE on a set of formal rewrite rules. By applying these rules, it is possible to derive how

$$
\begin{aligned}
[a_i \cdot b_j]_{(\sigma,G)} &\rightarrow [a_i \cdot b_j]_{(\sigma,G)} \\
[(a_i + b_j) \cdot \alpha]_{(\sigma,G)} &\rightarrow [(a_i \cdot \alpha + b_j \cdot \alpha)]_{(\sigma,G)} \\
[a_i \cdot b_j + a_i \cdot c_j]_{(\sigma,G)} &\rightarrow [(a_i \cdot (b_j + c_j)]_{(\sigma,G)} \\
[a_i + b_i]_{(\sigma,G)} &\rightarrow [t_i]_{(\sigma',G')} && t_i = \sigma[{}^{t_i'}/_{a_i + b_i}], G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, b_i)\}) \\
[(a_i \cdot b_j) \cdot \alpha]_{(\sigma,G)} &\rightarrow [t_i \cdot b_j]_{(\sigma',G')} && \sharp(b_j) > \sharp(a_i), t_i = \sigma[{}^{\sigma[\perp/a_i]}/_{a_i \cdot \alpha}], a_i \notin in(G), \\
& && G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, \alpha)\}) \\
[(a_i \cdot b_j) \cdot \alpha]_{(\sigma,G)} &\rightarrow [t_i \cdot b_j]_{(\sigma',G')} && \sharp(b_j) > \sharp(a_i), t_i = \sigma[{}^{t_i'}/_{a_i \cdot \alpha}], a_i \in in(G), \\
& && G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, \alpha)\})
\end{aligned}
$$

Fig. 5: Rewrite rules.

an expression will be transformed, as well as what and where (i.e. at which level in the loop nest) temporaries will be introduced. When applying a rule, the ER needs to update the state of the loop nest, to reflect, for example, the use of a new temporary and the newly created data dependencies. We define, therefore, the state of a loop nest $L = (\sigma, G)$, where $G = (V, E)$ represents the dependency graph, while σ maps invariant sub-expressions to identifiers of temporary arrays. We also introduce the *conditional hoister* operator [] on $\sigma : Inv \rightarrow S$ such that

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

That is, intuitively, if the invariant expression x has already been hoisted, then return the temporary identifiers that hosts its value; otherwise, hoist the expression. There is a special case when $v = \perp$, used to delete entries in σ. Specifically:

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

In other words, the previously hoisted expression x is removed (if any) and the temporary identifier that was hosting its value is returned. This is useful to express updates of invariant expressions. Rewrite rules for the ER are provided in Figure 5; obvious rules are omitted for brevity. Conceptually, the ER visits the expression tree from the root, which is the outermost operation, and applies the transformations dictated by the rewrite rules. As an example, one can try instantiating the rules in the code of Figures 2(a) and 3; eventually, the optimized code in Figures 2(b) and 4(b) is obtained, respectively.

... TODO... : each level in sigma is then taken and wrapped in a loop

...TODO...: to what extent applying rewrite rules, is problem specific...

...TODO...: things get interesting if the kernel operates on a batch of elements...can save a LOT of flops

```
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
  ...
  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],, A[i][j], ...)
      }
    }
  }
}
```

(a) Factorized, showing zeros, code

```
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<3; ++j) {
      for (int k=0; k<3; ++k) {
        M[j+3][k+3] += f(T1[j+3], A[i][k+3], ...);
        M[j][k] += g(T2[k], A[i][j], ...);
      }
    }
  }
}
```

(b) Factorized, skipping zeros, code

Fig. 6: Without and with skipping zeros

3.3. Avoiding Iteration on Zero-blocks by Symbolic Execution

The second task of the ER consists of skipping arithmetic operations over blocks of zero-valued entries in basis function and derivatives of basis functions arrays. Zero columns in such arrays arise, for example, when taking derivatives on a reference element and when employing mixed elements. In [Olgaard and Wells 2010], a code transformation to eliminate floating point operations on zero columns was introduced; the technique was implemented in FEniCS. We will evaluate our approach and compare it to this pioneering work in Section 5. We propose a strategy in which we avoid the use of indirection arrays in the generated code (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`), which otherwise would break the optimizations applicable at the Code Specializer level, especially SIMD vectorization.

Consider Figure 6(a), which is an enriched version of the transformed 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. We notice the presence of 4 zero-columns. Any multiplications or additions along these columns could (should) be skipped in order to minimize irrelevant floating point operations. One solution, as anticipated, is not to generate the zero columns, i.e. to generate a dense 6×2 array, to reduce the size of the iteration space over test and trial functions (from 6 to 2), and to use an indirection array (e.g. $ind = \{3, 5\}$) to access the right entries in the element tensor A. As explained, this prevents, among the various optimizations, effective code vectorizability, because memory loads and store would eventually reference non-contiguous locations.

Our approach is based on domain knowledge and symbolic execution. We discern the origin of zero-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 preserve the correctness of the element matrix evaluation while iterating along the space of test and trial functions. The two key observations are that 1) the number of zero-columns caused by mixed elements is much larger then that due to derivatives, and 2) such columns are contiguous in memory. Based on this, we decide to skip any computation involving zero-columns induced by mixed elements, whereas we still iterate over any other zero-columns. Consequently, the only modifications to the code that are eventually needed are adjusting loop bounds and introducing offsets when accessing the element matrix, as shown in Figure 6(b). In general, multiple iteration spaces over test and trial functions (i.e. `jk` loops), characterized by their own loop bounds, may be needed, increasing loop overhead and decreasing data locality for the element matrix.

The implementation is based on symbolic execution. The ER expects information from the higher layer about where the zero-blocks are located, in each tabulated basis function. This could come in the form of code annotation if the input to COFFEE were provided as a C string (as shown in Figure 6(b)) or by suitably decorating basis functions nodes in the abstract syntax tree representing the local assembly routine. Then, each statement in the local assembly loop nest is symbolically executed. For example, consider the statement `T2[r] = ((d*D[i][k])+(e*E[i][k]));` in Figure 6(b). Array `D` has non-zeros in positions $NZ_D = [3, 5]$; we also assume `E` has non-zeros in positions $NZ_E = [0, 2]$. Multiplications by scalar do not affect the propagation of zero columns. On the other hand, when summing the two operands `d*D[i][k]` and `e*E[i][k]`, we record that the target identifier `T2` will have non-zero columns in positions $NZ_D \cup NZ_E = [0 - 5]$. Exploiting the NZ information associated with each identifier, the assembly expression is split into several sets of sub-expressions, each set characterized by accesses to the same range of non-zero columns and, therefore, iterating over a suitably-sized iteration space. In our example, there is just one set, which contains two sub-expressions, enclosed in a doubly nested loop of size 3×3.

...TODO...: dire come si riaggiustano gli indici quando si vuole usare AVX ma la parte dense non e' multiplo del vector length - se non e' paddata la tail, allora si decrementa lo starting index di un pochino, tanto sono zeri...

4. CODE SPECIALIZATION

The Code Specializer is the second software component of COFFEE. It provides a range of code transformations tailored for optimizing 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 enrich this set of transformations as detailed in the following sections.

4.1. Vector-expansion of Invariant Terms

The Expression Rewriter's loop-invariant code motion hoists invariant sub-expressions in the body of the outermost loop along which they iterate. If such a sub-expression computes a value depending on the outermost loop only, then the only way of vectorizing it – if we exclude superword level parallelism [Larsen and Amarasinghe 2000], which is in general not applicable to our kernels – is by vector-expansion of terms.

An example is given in Figure 7(a): `F1, F2, F3, T1` are all scalar quantities, and their computation will not be auto-vectorized (as explained in Section 5, we experimented with the Intel and GNU compilers). The argument that the cost of evaluating such expressions is not relevant because their computational cost is $O(I)$, while the evaluation of the element matrix has a cost of $O(IT^2)$, is too superficial, in general. There are notable cases, for instance problems based on hyperelasticity, for which the

```
for (int i=0; i<I; ++i) {
  // Coefficient evaluation at a quadrature point
  for (int r=0; r<T; ++r) {
    F1 += (coeff[r][0]*A[i][r]);
    F2 += (coeff[r][0]*B[i][r]);
    F3 += (coeff[r][0]*C[i][r]);
  }

  // 1) Expression Rewriter's precomputed terms
  T1 = ((K0*F2)+(K1*F1)+(K2*F0)+1.0);
  ...
  for (int r=0; r<T; ++r) {
    // 2) Expression Rewriter's precomputed terms
    ...
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      // Evaluation of element matrix
    }
  }
}
```

(a) To vector expand code

```
// Vector-expanded code
for (int i=0; i<I; ++i) {
  // Coefficient evaluation at all quadrature points
  for (int r=0; r<T; ++r) {
    F1[i] += (coeff[r][0]*A[i][r]);
    F2[i] += (coeff[r][0]*B[i][r]);
    F3[i] += (coeff[r][0]*C[i][r]);
  }
  // 1) Expression Rewriter's precomputed terms at
  // all quadrature points
  T1[i] = ((K0*F2[i])+(K1*F1[i])+(K2*F0[i])+1.0);
  ...
}
for (int i=0; i<I; ++i) {
  for (int r=0; r<T; ++r) {
    // 2) Expression Rewriter's precomputed terms
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      // Evaluation of element matrix
    }
  }
}
```

(b) Vector-expanded code

Fig. 7: Before and after vector expansion

Expression Rewriter produces a significantly-large number of invariant scalar terms, which dramatically impact the execution time.

We have implemented a procedure that vector-expands scalar expressions, while lifting them outside of the loop nest as shown in Figure 7(b). The dependency graph is queried to assess the legality of the transformation. At the price of extra memory, vector-expanded terms can now be vectorized by the compiler.

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 matrices 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 calls. The study was conducted only in the case of a specific Helmholtz equation. There are cases, however, in which tabulated basis functions sizes grow up to a point for which certain BLAS implementations allow to achieve a better performance. For example, this may be the case of relatively-high polynomial orders or when the form is characterized by the presence of a number of pre-multiplying functions. To explore these extreme yet meaningful scenarios, we have developed an algorithm to reduce any generic element matrix evaluation to a sequence of DGEMM calls.

The algorithm is shown in Figure **??**. By using the rewrite rules in Figure 5, it can be proved that the Expression Rewriter can always reduce an assembly expression 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. Terms in the bodies of the various enclosing loops (e.g. coefficients evaluation at quadrature point, temporaries introduced by the Expression Rewriter) are vector-expanded and hoisted outside of the loop nest, as explained in Section 4.1. This implies that the loop nest is now perfect; that is, there is no intervening code among the various loops. The element matrix evaluation became a sequence of perfect loop nests, each loop nest computing a dense matrix-matrix multiply between temporaries hoisted by the Expression Rewriter or tabulated basis functions. 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 of each loop nest into a DGEMM call is the last, straightforward step.

4.3. Model-driven Dynamic Autotuning

We have demonstrated in [Luporini et al. 2014] that an optimal set of code transformations does not exist. What sequence of transformations applying to a problem to optimize its performance depends on 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 that COFFEE could use to choose, given a problem, the transformations that were likely to maximize the performance. By having added a significant number of options to the set of possible optimizations, the selection problem is now far more challenging. The sole Expression Rewriter, for instance, could apply rewrite rules up to a set of pre-established depths, leading to the generation of many possible code variants.

We tackle this problem by compiler autotuning. For each problem, not only does it allow to determine the best combination of transformations out of the set presented so far, also it enables exploring parametric low-level optimizations, such as loop unroll, unroll-and-jam, and interchange, by trying multiple unroll factors and loop permutations. By leveraging the cost model defined in our previous study, domain-awareness, 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 implementation. 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]).

The structure of the autotuner is outlined in Figure **??**. COFFEE analyzes the input problem and decides what variants it is worth testing, as described later. It then provides the autotuner with all of the possible variants, 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, repeatedly in a *while* loop executed a pre-established amount of time (order of milliseconds), 1) initializes kernel's input variables with fictitious values and 2) calls the kernel. At the exit of the *while* loop, the times the kernel was invoked is recorded. Eventually, the variant executing 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.

Also, the following heuristics, which capture properties of our computational domain, are used

— When both test and trial functions derive from the same function space, the lengths of their corresponding loops are identical. In this case, since for the employed storage layout the memory accesses are symmetrical along these two loops, their interchange is pruned from the search space.
— In general, if relatively high polynomial orders are employed, the loop nest is larger. In these cases, in order to avoid testing too many unroll factors, we impose a bound X on the loop nest's overall unroll factor, which we found empirically. For example, the sum of the unroll factors of all outer loops cannot exceed X.
— On the other hand, if the polynomial order is low, i.e. for smaller loop nests, we prune variants that we know will be low-performing, e.g. those resorting to BLAS.
— We select two levels of expression rewriting. In the "base" level, generalized loop-invariant code motion only is applied. This means that only a subset of the rewrite rules exposed in 5 are applicable. In the "aggressive" level, all of the rewrite rules are applied.
— For the expression splitting optimization described in [Luporini et al. 2014] and summarized in Section 2.2, we test only only three split factors, namely 1, 2, 4. Also, if the input problem uses mixed function spaces, the iteration space is already split by the Expression Rewriter to avoid computation over zero-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.

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`.

...TODO... Problems chosen...

...TODO... We don't compare to tensor contraction by transitivity on olegaards work

5.2. Results for Forms of Increasing Complexity

In Figure **??**, ...

6. CONCLUSIONS

REFERENCES

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

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

Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 145–156. DOI:http://dx.doi.org/10.1145/349299.349320

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