A

Optimizing Finite Element Integration through Automated 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 Finite Element Integration through Automated Expression Rewriting and Code Specialization. *ACM Trans. Arch. & Code Opt.* V, N, Article A (January YYYY), 6 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

2. PRELIMINARIES

2.1. Quadrature for Finite Element Local Assembly

2.2. Code Generation for Quadrature Representation

Rapid implementation of high performance, robust, and portable code evaluating element matrices using quadrature can be achieved through automated code generation. This has been succesfully proved in the context of the popular FEniCS project [Logg et al. 2012]. The FEniCS Form Compiler (FFC) accepts as input the variational form of a partial differential equation and generates C++ code implementing local assembly routines. The variational form is expressed at high-level by means of the domain-specific Unified Form Language (UFL). Local assembly code must be high performance: as the complexity of a form increases, in terms of number of derivatives, pre-multiplying functions, and polynomial order of the chosen functions, the resulting ker-

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

nels evaluating element matrices become more computationally expensive, which impacts significantly the run-time of the overall computation.

Achieving high performance is non-trivial due to the complexity of the mathematical expressions involved in the numerical integration and because of the small sizes of loops and accessed arrays. In [Olgaard and Wells 2010], [Kirby et al. 2005] and [Russell and Kelly 2013], it is shown how automated code generation allows the introduction of powerful optimizations, which a user cannot be expected to write "by hand", as well as the exploration of non-standard integration techniques, based, for instance, on symbolic manipulation. In [**?**] we made one step forward by showing that different problems require distinct set of transformations if close-to-peak performance needs to be obtained, and that low-level, domain-aware code transformations, which are not supported by available compilers, are essential to maximize instruction-level parallelism and register locality.

2.3. Summary of Low-level Optimizations for Quadrature Representation

To neatly distinguish the contributions of this paper from those in [**?**], in this section we summarize the results of our previous work on automated code transformations for quadrature representation.

...TODO...

The work in [**?**] resulted in the development of COFFEE[1], a compiler for the optimization of local assembly kernels relying on quadrature representation.

...TODO...

Temporary arrays can be placed at the right depth in the sorrounding loop nest to store values of sub-expressions that are invariant to one or more loops.

...TODO...

3. A COMPILER FOR OPTIMIZING QUADRATURE-BASED INTEGRATION

In order to generate high performance code, mathematical expressions evaluating the element tensor must be optimized with regards to several interrelated aspects: 1) minimization of floating point operations, 2) instruction-level parallelism, and 3) data locality. In this paper, we tackle these three points building on our previous work ([**?**]).

...We propose a novel structure of how we think a platform-independent, domain-specific optimizing compiler should look like....

...The Expression Rewriter is a software module implemented in COFFEE dealing with the first point...

...The Code Specializer (henceforth CS) is... A key point is that the ER has to perform transformations that do not break the code specializer. Having indirections is really dangerous...

4. EXPRESSION REWRITING

As summarized in 2.3, loop-invariant code motion is the key to reduce the computational intensity of a mathematical 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

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

```
for (int i=0; i<I; ++i)
  for (int j=0; j<T; ++j)
    for (int k=0; k<T; ++k)
      A[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]
```

Fig. 1: Original code

operations in an expression. In Section 4.1, 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 4.3, we show a novel approach to avoiding computation on zeros based on symbolic execution.

4.1. Objectives of the Expression Rewriter

Consider the simplified example of the element matrix computation in Figure **??**, 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 [**?**], 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 **??**, 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 **??**. 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

```
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) {
      A[j][k] += ((T1[j]*A[i][k]*g)+((A[i][k]/c)+(T2[k]*f))*A[i][j])*det*W[i];
    }
  }
}
```

Fig. 2: Factorized 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]));
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      A[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

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 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 **??**. 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.

```
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]))*detW[i];
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      A[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]))*detW[i];
    T2[r] = T1[r]*g;
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      A[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.

4.2. 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 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

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

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 **??** and **??**; eventually, the optimized code in Figures **??** and 4(b) is obtained, respectively.

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

Fig. 5: Rewrite rules.

REFERENCES

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

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

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

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).