Optimized Code Generation for Finite Element Local Assembly Using Symbolic Manipulation

FRANCIS P. RUSSELL, Imperial College London
PAUL H. J. KELLY, Imperial College London

Automated code generators for finite element local assembly have facilitated exploration of alternative implementation strategies within generated code. However, even for a theoretical performance indicator such as operation count, an optimal strategy for local assembly is unknown. We explore a code generation strategy based on symbolic integration and polynomial common sub-expression elimination (CSE). We present our implementation of a local assembly code generator using these techniques. We systematically evaluate the approach, measuring operation count, execution time and numerical error using a benchmark suite of synthetic variational forms, comparing against the FEniCS Form Compiler (FFC). Our benchmark forms span complexities chosen to expose the performance characteristics of different code generation approaches. We show that it is possible with additional computational cost, to consistently achieve much of, and sometimes substantially exceed, the performance of alternative approaches without compromising precision. Although the approach of using symbolic integration and CSE for optimizing local assembly is not new, we distinguish our work through our strategies for maintaining numerical precision and detecting common sub-expressions. We discuss the benefits of the symbolic approach for inferring numerical relationships, and analyze the relationship to other proposed techniques which also have greater computational complexity than those of FFC.

Categories and Subject Descriptors: G.1.8 [**Numerical Analysis**]: Partial Differential Equations – Finite element methods; G.4 [**Mathematical Software**]: Efficiency, reliability and robustness; I.1.4 [**Symbolic and Algebraic Manipulation**]: Applications

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Finite element method, code generation, common sub-expression elimination.

ACM Reference Format:
Russell, F. P. and Kelly, P. H. J. 201Y. Optimized Code Generation for Finite Element Local Assembly Using Symbolic Manipulation ACM Trans. Math. Softw. V, N, Article A (201Y), 30 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The evaluation of local element matrices is of key importance in the implementation of the finite element method. These matrices must typically be evaluated for every cell in the discretized domain, so it is important that their calculation is efficient if high performance is to be achieved. However, the process of evaluating these matrices

This work was supported by EPSRC under grant EP/I00677X/1 and a doctoral training studentship. The authors would like to acknowledge Florian Rathgeber and Chris Cantwell for their insightful feedback.
Author's addresses: F. P. Russell & P. H. J. Kelly, Department of Computing, Imperial College London.

also has a direct correspondence to a much higher level mathematical description, the computation of an integral of a variational form over a subset of the domain.

The need to be able to produce efficient local assembly implementations, while also allowing computational scientists to vary the associated variational forms has led to research into automated code generators. One such generator is the FEniCS Form Compiler [Kirby and Logg 2006] (FFC) which is being developed as part of the FEniCS project. FFC takes as input the Unified Form Language (UFL), a domain specific language that describes the variational form to be integrated. The output is a C++ implementation conforming to the Unified Form-assembly Code (UFC) specification [Alnæs et al. 2009], an interface defined by the FEniCS project that specifies how different components of a finite element assembly implementation interact.

High level languages for describing variational forms such as UFL allow the specification of assembly to be abstracted from its implementation. As such, it facilitates code generators to vary aspects of the generated code in a way that would be impossible to maintain in any hand written implementation.

The traditional approach to assembly has been to evaluate integrals using numerical quadrature. Quadrature based assembly implementations are relatively simple to write and have well understood performance characteristics, both desirable aspects for any implementation. Research into other implementation choices has led to the development of a *tensor contraction representation* [Kirby and Logg 2006], which represents the process of local assembly a contraction between cell geometry dependent and independent tensors.

The tensor contraction representation of assembly has been shown to be significantly more efficient for evaluating certain classes of variational forms when compared to quadrature based assembly [Kirby and Logg 2006]. An extension to this technique uses topological relationships between the entries in the geometry independent tensor to find redundancies in the computation, requiring more extensive analysis of the form being evaluated [Kirby et al. 2006]. Such an approach is an example of the efficiency improvements that can only be obtained when using automated code generation.

Work by Ølgaard and Wells [Ølgaard and Wells 2010] has shown that for some variational forms, the tensor contraction representation of assembly outperforms quadrature by a factor of 300 in terms of operation count. However, for other forms, quadrature representation can outperform tensor contraction by a factor of 100. Kirby et al. state that the algorithm they use for reducing the operation count of the tensor contraction is optimal with respect to the optimizations they can express [Kirby et al. 2006]. This suggests that for some classes of forms, it is an inherent property of the topological optimization technique which prevents it from reaching the efficiency of quadrature for certain classes of forms.

One of our motivations for this work is to determine that if provided with a sufficiently expressive representation of local assembly, a system can generate code that equals or surpasses the efficiency of both tensor contraction and quadrature implementations. We observe that both quadrature and tensor contraction representations correspond to specific factorizations of the expressions required to evaluate local assembly matrices. If a code synthesis system is provided with a representation capable of expressing a superset of the factorizations corresponding to quadrature or tensor contraction based assembly, it should be possible for that system to generate code that is at least as efficient as both, at least in terms of operation count. As such a representation would be extremely flexible, the concern then becomes how to traverse the search space of possible local assembly implementations.

For our work, we have chosen to explore local assembly optimizations using symbolic algebra. We have implemented these optimizations in a C++ finite element library we have developed called EXCAFÉ [1]. EXCAFÉ has been designed to explore optimizations within the finite element method by capturing different aspects of the method using various domain-specific abstractions. For the purposes of this paper, we only consider the local assembly analyses implemented in EXCAFÉ.

Our approach can be summarised as follows:

(1) Compute symbolic representations for each scalar element of the local assembly matrix.
(2) Apply an optimization pass to these expressions that exploits common sub-expressions and other redundancies in order to compute an efficient evaluation strategy.

The more general aspects of this approach are not new. The FINGER system [Wang 1986], uses MACSYMA [Fateman 1989] to perform symbolic integration, followed by a CSE (common sub-expression elimination) pass, performed by REDUCE. More recent work on the SyFi Form Compiler [Alnæs and Mardal 2010], explores performing both analytical and quadrature based integration on symbolic expressions for local assembly matrix entries before applying a CSE pass.

Symbolically integrating the expressions representing a local assembly matrix at code-generation time means we avoid the use of quadrature at run-time. The tensor contraction scheme also avoids run-time quadrature by moving the cell integral into the expression for the reference tensor (for affine elements), which is computed at code-generation time. One can think of this a technique that reduces operation count by performing compile-time partial evaluation of integrals. However, the symbolic integration scheme also destroys the loop structure of the computation that tensor contraction and quadrature based code generation schemes exploit. Reclaiming such structure is vital if the resulting code is to be efficient.

We distinguish ourselves from previous work through the efforts we take to construct efficient code from the resulting symbolically integrated expressions. We use existing work by Hosangadi et al. [Hosangadi et al. 2006] on polynomial common sub-expression elimination (CSE) to find factorizations that exploit the distributivity of multiplication over addition. We extend this work to take account of numerical relationships that would be opaque to the original Hosangadi et al. algorithms. Through these analyses we aim to generate superior code than that resulting from the conventional CSE passes that have been applied in generators such as SyFi [Alnæs and Mardal 2010].

Alnæs et al. demonstrate significant performance improvements [Alnæs and Mardal 2010] when comparing code generated by SyFi to other hand written finite element implementations such as Diffpack [Bruaset and Langtangen 1997] and deal.II [Bangerth et al. 2007]. However, they do not perform an extensive comparison against the quadrature and tensor contraction implementations generated by FFC. We believe that comparisons against other finite element implementations that do not use automated code generation primarily provide evidence for the benefits of code generation, and not for the symbolic manipulation approach in particular. Therefore, we only compare against the FFC generated quadrature and tensor contraction implementations. One of our motivations in this work is to determine if symbolic techniques can be used to achieve comparable or improved performance over both these implementation choices.

[1]Available at `http://www.doc.ic.ac.uk/~fpr02/excafe/` and also supplied as auxiliary material to this publication.

2. PRELIMINARIES

We consider the weak formulation of a general linear variational problem as follows. Find $u \in \mathcal{X}$ such that

$$a(u, v) = L(v),\ \forall v \in \mathcal{V} \tag{1}$$

where a and L are bilinear and linear forms respectively. The function spaces $\mathcal{X}$ and $\mathcal{V}$ are called the trial and test spaces, respectively. They both contain an infinite set of functions so we approximate them with the discrete function spaces $\mathcal{X}^\delta$ and $\mathcal{V}^\delta$ using the basis function sets Φ and Ψ. Our problem now reads, find $u^\delta \in \mathcal{X}^\delta$ such that

$$a(u^\delta, v^\delta) = L(v^\delta),\ \forall v^\delta \in \mathcal{V}^\delta \tag{2}$$

Our solution field u^δ is expressed as a linear combination of test functions:

$$u^\delta(x) = \sum_{j=0}^{|\Phi|} \hat{u}_j \Phi_j(x) \tag{3}$$

We can now define our linear system as follows:

$$A\mathbf{x} = b \tag{4}$$

where A and b are the discretized versions of a and L respectively, defined as a matrix and vector:

$$A_{ij} = a(\Phi_j(x), \Psi_i(x)) \tag{5}$$
$$b_i = L(\Psi_i(x)) \tag{6}$$

and the unknown x is a vector containing the basis function coefficients of the solution field so that:

$$\mathbf{x}_j = \hat{u}_j \tag{7}$$

The domain Ω over which the partial differential equation is solved is partitioned into cells. The basis functions are defined so that they are only non-zero valued on neighbouring cells. As a consequence, the matrix A is sparse. Typically, the basis functions are defined over a reference *reference cell*, and the process of performing a co-ordinate transformation to the cell being integrated over is incorporated in the assembly process.

We assume that our domain Ω is partitioned into a set of cells, K. We define the following:

$\chi^k(\xi)$, a function that transforms a local co-ordinate ξ on the reference cell to the global co-ordinate x on the cell k.

$\iota^k(i)$, a function that returns the global numbering of the local basis function i defined on cell k.

ϕ and ψ, the local versions of the basis function sets Φ and Ψ respectively. For all local basis functions $1 \leq p \leq |\phi|$, $1 \leq q \leq |\psi|$ on any cell $k \in K$:

$$\Phi_{\iota^k(p)}(\chi^k(\xi)) = \phi_p(\xi) \tag{8}$$
$$\Psi_{\iota^k(q)}(\chi^k(\xi)) = \psi_q(\xi) \tag{9}$$

We can now define our local assembly matrix for cell k as follows:

$$M^k_{qp} = a(\Phi_{\iota^k(p)}, \Psi_{\iota^k(q)}) \tag{10}$$

The functional a corresponds to an integral over cell k. However, via a change of variables, it can be rewritten as an integral over a reference region Ω_{st}. The integration is then performed with respect to local co-ordinates and uses the local basis sets ϕ and ψ instead of the global ones Φ and Ψ.

The change of variables requires multiplying the integrand by $|J(\chi^k(\xi))|$. This term is the determinant of the Jacobian of the co-ordinate transformation from local to global co-ordinates, and acts as a scaling factor.

a may also contain derivatives of the basis functions w.r.t. global co-ordinates. We can compute any global derivatives from local ones by application of the chain rule. In particular, we transform the gradient of a function defined in local co-ordinates to the corresponding global gradient by taking the inner product between inverse of the gradient of local-to-global co-ordinate transformation χ^k, and the gradient of the function w.r.t. local co-ordinates. Let us assume we have some function f defined in terms of the local co-ordinate space ξ and we wish to find the gradient of F_k, the function f transformed to cell k and expressed in terms of global co-ordinates

$$\nabla F_k(x) = (\nabla \chi^k(\xi))^{-1} \cdot \nabla f(\xi) \tag{11}$$

where

$$x = \chi^k(\xi) \tag{12}$$

The presence of derivatives in the variational forms being integrated has important performance implications for the tensor contraction representation of assembly as discussed in the next section.

3. CURRENT APPROACHES TO LOCAL ASSEMBLY

In this section, we briefly review the quadrature and tensor contraction based approaches to performing local assembly. Both these approaches can be described in a succinct mathematical form. In contrast, the symbolic approach permits more arbitrary optimizations and therefore tends to produce unstructured results. The topological optimizations to the tensor contraction representation [Kirby et al. 2006] have the same effect, as they also result in extremely form-specific optimizations.

As an example, we consider how to evaluate the local assembly matrix for the Laplacian operator:

$$a(u, v) = \int_\Omega \nabla u(x) \cdot \nabla v(x)\, dx \tag{13}$$

3.1. Quadrature

The conventional approach to local assembly is via quadrature [Sherwin and Karniadakis 2005]. The integral is performed by evaluating the function to be integrated at specific points of the cell, and taking a weighted sum of these values. For our Laplacian example, we can write this as follows:

$$M^k_{qp} = \sum_{i=0}^{Q-1} w_i (\nabla \chi^k)^{-1} \cdot \nabla \phi_p \cdot (\nabla \chi^k)^{-1} \cdot \nabla \psi_q |J(\chi^k)| \tag{14}$$

The quadrature points and weights are typically chosen using Gaussian quadrature rules. An m point rule will exactly integrate univariate polynomials up to degree $2m - 1$. Higher-dimensional quadrature rules for simplices and hypercubes can be derived from the one-dimensional variants. Gaussian quadrature does not use points at either end of the interval being integrated, but there are also variants known as Gauss-Radau and Gauss-Lobatto, which define points at one and both ends, respectively.

A typical implementation will consist of a loop that iterates over the Q quadrature points, and sums into the local assembly matrix. The basis functions and their derivatives (if needed) at the quadrature points can be computed just once and reused, since they are only dependent on local co-ordinates.

For affine mappings, both the determinant of the Jacobian of the co-ordinate transformation, $|J(\chi^k)|$, and the inverse gradient of the co-ordinate transformation, $(\nabla\chi^k)^{-1}$, only need be computed once since they will remain constant across the cell. If the elements are not affine, they will need to be recomputed at each quadrature point.

The FEniCS form compiler also performs other optimizations for for quadrature based code generation [Ølgaard and Wells 2010]. Firstly, if a basis function or its derivatives evaluate to zero at all quadrature points, operations on those values can be eliminated. Secondly, loop invariant code motion is used to avoid recomputing values that are independent of the values of the quadrature summation indices.

3.2. Tensor Contraction

The tensor contraction representation is a local assembly implementation choice investigated as part of the FEniCS project, and implemented in FFC. It works by expressing the local assembly matrix as a tensor contraction between a geometry independent *reference tensor* A^0 and a *geometry tensor* G_k. The geometry tensor is cell-dependent and must be recomputed for each cell. The reference tensor is independent of cell geometry and only needs to be computed once, during the code generation phase. The local assembly matrix can be written as follows:

$$M^k = A^0 : G_k \tag{15}$$

where the : operator represents a summation over zero or more indices. In the case of our Laplacian operator, the geometry and reference tensors are defined as follows:

$$A^0_{qp\alpha\beta} = \int_{\Omega_{st}} \frac{\partial\phi_p}{\partial\xi_\alpha}\frac{\partial\psi_q}{\partial\xi_\beta}\, d\xi \tag{16}$$

$$G_k^{\alpha\beta} = |J(\chi^k)| \sum_{\gamma=0}^{d} \frac{\partial\xi_\alpha}{\partial x_\gamma}\frac{\partial\xi_\beta}{\partial x_\gamma} \tag{17}$$

Here, α, β and γ are co-ordinate directions that we take partial derivatives with respect to. Our trial and test basis function numberings are p and q respectively. The chain rule has been applied to the partial derivatives of the Laplacian and the geometry-independent terms moved into the reference tensor.

The per-cell cost of assembly is the number of operations required to evaluate the geometry tensor and to perform the contraction between the reference and geometry tensors. In some cases, this is a significant reduction in operations over a quadrature based implementation.

For affine mappings, the integral over the reference cell is only present in the definition of the reference tensor. Since the reference tensor is evaluated at code-generation time, the run-time cost of quadrature is not present in the tensor contraction representation. As a consequence, tensor contraction representation tends to perform well for higher order forms that would otherwise require a large number of quadrature points to evaluate.

Kirby and Logg describe how the tensor contraction representation may be extended to non-affine mappings in [Kirby and Logg 2006], Section 3.4. In the case of curvilinear elements, the integral cannot be evaluated at code-generation time. As a consequence, both the geometry and reference tensors have one extra dimension corresponding to an index for quadrature points.

3.2.1. Graph-Based Optimizations. Further work on improving the performance of tensor contraction representation has looked at reducing the operation count required to perform the contraction between the geometry and reference tensors.

Each element of the local assembly matrix can be considered an inner product between a vector of values from the reference tensor and the entire geometry tensor. The values of the elements of the reference tensor are known at code-generation time, but the values of the geometry tensor are only known at run-time.

Kirby et al. have used metrics based on Hamming distance and collinearity between vectors to characterise the number of multiply-add pairs (MAPs) required to compute one inner product from another. Using these, they develop a code generation scheme that is optimal with respect to the number of MAPs required to compute a local assembly tensor exploiting these relationships [Kirby et al. 2006]. Kirby et al. call these *topological optimizations*.

Kirby and Scott have extended this work to exploit linear dependence between vectors in the reference tensor [Kirby and Scott 2007]. The presence of linear dependence relationships between vectors in the reference tensor enables inner products to be computed from weighted sums of other inner products. This can be more efficient than computing the inner products outright. Kirby and Scott call these *geometric optimizations* and show instances where they reduce operation count over topological optimization techniques.

In terms of the reduction of operation count expressible within each framework, the topological optimizations are optimal whereas the geometric optimizations are not. The fact that the geometric optimizations can improve upon the topological ones indicate that the latter only achieves this optimality result due to the restricted nature of the expressible optimizations.

Wolf and Heath have also looked at exploiting linear dependencies between vectors in the reference tensor although they limit themselves to relationships involving three vectors (*coplanar* relationships) [Wolf and Heath 2009]. However, they generalise the Hamming distance and collinearity relationships to the *partial collinearity* relationship. Code exploiting this relationship makes use of collinearity between a subset of the elements of two vectors from the reference tensor to reduce its operation count. Hence, this work uses both topological and geometric techniques to reduce its operation count.

We describe these optimizations assuming vectors of values from the reference tensor a_0, a_1 and a_2 and the geometry tensor flattened to a vector, g.

Hamming Distance. This optimization generates code that derives some inner product $a_1 \cdot g$ from $a_0 \cdot g$. If $a_0 \cdot g$ is already known, then $a_1 \cdot g$ can be calculated as $(a_1 - a_0) \cdot g + a_0 \cdot g$.
If a_0 and a_1 have many entries in common, then the vector $a_1 - a_0$ will contain many zeros. Hence, $(a_1 - a_0) \cdot g$ can be computed with a cost proportional to the number of non-zero values in $a_1 - a_0$ rather than the length of the vectors. Therefore, it may require fewer operations to compute $(a_1 - a_0) \cdot g$ and add it to the already known value $a_0 \cdot g$, compared to computing $a_1 \cdot g$ directly.

The cost of deriving $a_1 \cdot g$ from $a_0 \cdot g$ increases for each entry of a_0 and a_1 that is different. Hence, the cost is proportional to the Hamming distance between the two vectors.

Collinearity. This optimization derives an inner product $a_1 \cdot g$ from $a_0 \cdot g$ where $a_1 = \alpha a_0$. This requires that both vectors from the reference tensor are scaled versions of each other. Computing a value in this way only requires a single multiplication by the scaling factor α.

Partial Collinearity. This optimization generalises the Hamming distance and collinearity optimizations previously described. Whereas the Hamming distance optimization requires the two vectors a_0 and a_1 to have elements in common, the partial collinearity optimization takes advantage of corresponding elements in a_0 and a_1 when related by a common scaling factor.

Coplanarity and Linear Dependence. If multiple vectors in the reference tensor are linearly dependent, it is possible to calculate an inner product using a weighted combination of other inner products.

For example, if a_0, a_1 and a_2 are linearly dependent, $a_1 \cdot g$ can be computed as $\alpha a_0 \cdot g + \beta a_2 \cdot g$ for some α and β.

If the set of linearly dependent vectors has two elements, this is again the collinearity relationship. If it is of size three, the relationship is coplanar.

Kirby et al. have implemented the Hamming distance and collinearity optimizations in the FErari library which is used by FFC when it generates optimized tensor contraction implementations.

FErari builds a total undirected graph over the entries of the local assembly matrix, where each edge is weighted with the number of MAPs required to compute one entry from the other. A minimal spanning tree is constructed over this graph in order to determine the execution strategy with the fewest number of MAPs. The resulting code is optimal in the sense it requires the fewest number of operations with respect to the redundancies FErari can exploit [Kirby et al. 2006].

4. CODE GENERATION IN EXCAFÉ

In this section, we describe the implementation of the local assembly code generator implemented in EXCAFÉ. Specification of variational forms is done inside the library, using C++ function calls and operator overloading in a syntax similar to UFL. We provide in Figure 1 a simple comparison of UFL and our C++ Domain Specific Language (DSL) when used to specify the Laplacian operator. In order to facilitate comparisons with FFC generated code, we have modified EXCAFÉ to output code conforming to the UFL `cell_integral` C++ class interface.

4.1. Overview

The process of code generation in EXCAFÉ for a given variational form consists of the following steps:

(1) Compute symbolic representations of the basis functions used by the form to be compiled (Section 4.2).
(2) Symbolically compute the *integrand* of the variational form for each choice of trial and test function (Section 4.3).
The result is a matrix of symbolic expressions in which cell geometry, cell-local co-ordinates and basis function coefficients are unknowns.
(3) Analytically integrate each expression in the matrix computed in the previous step over the reference cell (Section 4.4).

```
element = FiniteElement("Lagrange", triangle, 2)

v = TestFunction(element)
u = TrialFunction(element)

a = dot(grad(u), grad(v))*dx
```

(a) UFL

```
static const std::size_t dimension = 2;
Scenario<dimension>& scenario = getScenario();

Element element = scenario.addElement(new LagrangeTriangle<0>(2));
BilinearFormIntegralSum a = B(grad(element), grad(element))*dx;
```

(b) EXCAFÉ

Fig. 1: Specification of the Laplacian operator in UFL and EXCAFÉ. The EXCAFÉ example is not self-contained, since it assumes the existence of a `Scenario` object, which contains problem context information such as the mesh. The template parameter to `LagrangeTriangle` is the rank of the basis, in this case zero, since it is a scalar field.

The result is matrix of expressions in which cell-local co-ordinates are no longer unknowns (since they have been removed via integration). Code generation could be performed at this step, but the result could contain many redundancies.

(4) Apply common sub-expression elimination to the symbolic expressions of the matrix computed in the previous step to find an efficient evaluation strategy (Section 5).

(5) Generate a UFC [Alnæs et al. 2009] compliant cell integration function. We note that EXCAFÉ was not designed as a form compiler, and we generate UFC compliant code primarily to facilitate comparisons against FFC.

We describe the symbolic manipulation EXCAFÉ performs in the rest of this section, and the common sub-expression elimination technique we apply in Section 5.

4.2. Basis Functions

Currently, only the Lagrange basis functions over triangles have been implemented in EXCAFÉ, but the techniques we describe easily generalise to any polynomial basis set over simplex or hypercube elements.

Construction of the Lagrange basis functions is implemented in EXCAFÉ using the techniques implemented in FIAT [Kirby 2004]. An orthogonal basis (called the *prime basis*) is first constructed. Next, a matrix similar to a Vandermonde matrix is inverted in order to find the linear combination of prime bases that form a nodal basis set. For our triangular elements, the prime basis is the Dubiner orthogonal basis [Dubiner 1991], derived from the Jacobi polynomials.

In EXCAFÉ, construction of the Jacobi polynomials and the matrix inversion step are performed entirely using rational numbers. Hence, our symbolic representation of the Lagrange basis functions is exact, and not subject to any form of floating point related inaccuracy. Keeping the coefficients in our symbolic representations as rational numbers wherever possible, rather than floating point values, is particularly important for our factorization optimizations, described in Section 5.

4.3. Variational Forms

Variational forms typically consist of vector calculus and tensor operators applied to basis functions, and discretized fields represented as a linear combination of basis functions.

Tensor operators can be applied directly to the symbolic expressions. The differential operators require the application of the chain rule in order to correctly transform derivatives.

The local-to-global co-ordinate mapping is represented using the basis functions defined over the reference cell. Since we only handle affine transformations, we use the linear Lagrange basis functions to transform from local to global co-ordinates.

Symbolically taking the gradient of the co-ordinate transformation and applying Cramer's rule to invert it allows us to define expressions for the entries of the tensor $(\nabla \chi^k(\xi))^{-1}$, which we use to transform a locally defined gradient to a global one on the arbitrary cell k.

Similarly, it is possible to compute the local-to-global scaling factor, $|J(\chi^k)|$ analytically using the same techniques. We note that in a strict mathematical derivation, it is possible for the local-to-global scaling factor to become negative if the cell orientation has been reversed. The UFC [Alnæs et al. 2009] specification defines cell-local vertex numbering in terms of global vertex numbering, which does not easily permit the preservation of cell orientation. Therefore, we take the modulus of the scaling factor instead, which allows cell orientation to be reversed without negating the value of the cell integral. Code generated by the FEniCS Form Compiler does this also.

4.4. Symbolic Integration

We eliminate any need for run-time quadrature by symbolically integrating the expression for each element of the local assembly matrix. To integrate over the reference triangle, we first perform a co-ordinate transformation to the bi-unit square. We can then integrate over each dimension individually.

For certain classes of forms, symbolic integration can be particularly computationally expensive. We could use quadrature at code-generation time, however, this would introduce floating point values into our symbolic expressions, limiting the effectiveness of our factorizer.

Our original symbolic representation used the C++ computer algebra library GiNaC [Bauer et al. 2002] as a back-end. However, we later implemented our own symbolic representation to explore implementation choices related to symbolic integration and selective expansion of certain terms. Since we have only considered affine mappings, the expressions we integrate are always polynomial in the variables we integrate over (the cell-local co-ordinates).

Pre-multiplication by multiple co-efficient functions results in expressions that have a product-of-summations structure (since each field is represented by a weighted sum of basis functions). This appears to be the primary source of inefficiency for the class of forms we have chosen to benchmark over. Expanding complex products before integration can result in extremely large numbers of terms. However, performing integration before expansion typically requires integration by parts, which can also lead to large increases in the integrated expression size.

We have adopted the strategy of performing the minimal expansion of the input expression that also permits trivial integration of the resulting terms. Specifically, before integrating, we rewrite the integrand as an expanded polynomial w.r.t. the integration variable. This strategy appears to offer acceptable performance characteristics for the classes of forms we have evaluated. In particular, it avoids integration by parts and only expands the sub-terms of the input expression necessary to permit trivial integration of the resulting terms.

An alternative strategy would be for us to interface with a computer algebra system such as MAXIMA (an open-source fork of MACSYMA [Fateman 1989]) or similar system that uses pattern matching of mathematical expressions to improve symbolic integration performance [Slagle 1963; Harrington 1979]. It is unclear how difficult this would be to implement.

Since the symbolic integration pass typically results in extremely complex expressions, we choose to distribute sums over products in order to permit simplification by summing identical terms with different coefficients. After this step, each entry of the local assembly matrix is represented by a sum of rational expressions.

5. POLYNOMIAL FACTORISATION AND COMMON SUB-EXPRESSION ELIMINATION

Having constructed symbolic expressions for each entry of the local assembly matrix, it is necessary to find an efficient execution strategy for evaluating them. Approaches in previous work [Wang 1986; Alnæs and Mardal 2010] have primarily consisted of the application of standard compiler common sub-expression elimination (CSE) passes.

After the application of symbolic integration and expression normalisation, the resulting expressions bear little resemblance to the original input. Tensor contraction and quadrature implementations make use of the structure of the input form to product efficient code. Since our transformations destroy that structure, it is important that we use a CSE pass capable of recovering enough structure to produce code that is competitive with those implementations.

Our factorizer is based on the work of Hosangadi et al. on factorizing sets of multivariate polynomial expressions [Hosangadi et al. 2006]. Most importantly, it is capable of finding factorizations that employ distributivity. We have extended this work further with improvements to the types of factorizations that can be detected.

The Hosangadi et al. algorithm cannot be applied directly to our expressions. The presence of division (used in computing the inverse of the determinant of the coordinate transformation) and the modulus operator means that our expressions are not simply multivariate polynomials. To work around this, we replace non-polynomial expressions and expressions raised to negative exponents with temporary variables. We also extract any polynomial sub-expressions contained in non-polynomial ones.

Our input to the factorizer is a set of all polynomial sub-expressions present in the expanded form of the original expressions. We do however, permit negative exponents to remain in the products. The term x^{-n} can be treated identically to y^n where y corresponds to x^{-1}. This is particularly useful for our extension for handling numerical relationships between coefficients, described later.

5.1. Hosangadi et al. Factorisation Algorithm

We first provide a brief overview of the Hosangadi et al. algorithm. For a full description, consult [Hosangadi et al. 2006]. Hosangadi et al. use the following definitions:

Literal. A constant value or variable (e.g. 4.5 or x).
Cube. A product of literals raised to non-negative integer exponents. Cubes also have positive or negative signs associated with them (e.g. $7xy^2$ or $-2.5a^3b^2$).
SOP. A "sum of products" representation of a polynomial, consisting of a sum of cubes (e.g. $5.4x^2y + 2xy + 7$).
Cube-free. A SOP expression is "cube-free" if the only cube that can divide every cube in the SOP is the literal "1". For example, the SOP $a^2b + cd$ is cube free since a^2 and cd have no common divisors.

Kernel. For some SOP P and a cube c, the expression P/c is a kernel if it is cube-free and it has at least two terms. For example, letting $P = a^2bc + ac$, makes $P/ac = ab + 1$ a kernel.
Co-Kernel. This name is given to the cube dividing the SOP in a kernel expression. In the above example, this is ac.

A kernel expression represents a factorization of an SOP, P, of the form $P = C * F_1 + F_2$ where C is a cube and F_1 and F_2 are SOPs. For example, consider the factorization of the SOP $a^2b^2 + ab + ac$:

$$P = a^2b^2 + ab + ac \tag{18a}$$
$$P/ab = ab + 1 \tag{18b}$$

This represents a factorization of P of the form $P = C * F_1 + F_2$ where:

$$C = ab \tag{19a}$$
$$F_1 = ab + 1 \tag{19b}$$
$$F_2 = ac \tag{19c}$$

The first step of the Hosangadi et al. algorithm is to construct a set of kernel expressions for each SOP in the input. For conciseness, we do not describe the kernel extraction algorithm here, it is detailed in [Hosangadi et al. 2006]. The algorithm computes the set of all possible kernel expressions for each SOP. Hosangadi et al. show that all minimal algebraic factorizations of a polynomial expression can be obtained from the set of kernels and co-kernels of an expressions. In this context, “minimal” refers to the property that the only common divisor of the products forming F_1 is “1”.

We consider a worked example on the following two input expressions. The bracketed subscripts denote term numberings.

$$a^2{}_{(1)} + ab_{(2)} + c_{(3)} \tag{20a}$$
$$ab_{(4)} + b^2{}_{(5)} + bc_{(6)} \tag{20b}$$

The algorithm computes all kernels and co-kernels of the input SOPs, giving:

$$(a^2 + ab + c)/1 = a^2 + ab + c \tag{21a}$$
$$(a^2 + ab + c)/a = a + b \tag{21b}$$
$$(ab + b^2 + bc)/1 = ab + b^2 + bc \tag{21c}$$
$$(ab + b^2 + bc)/b = a + b + c \tag{21d}$$
$$\tag{21e}$$

The next step of the algorithm uses the kernel/co-kernel decomposition to find a factorization of the input SOPs. The kernels and co-kernels are written in the form of a matrix as shown in Figure 2. This matrix is called the Kernel Co-kernel matrix (KCM).

The rows of the matrix represent co-kernels and may be repeated if the same co-kernels are derived from different input expressions. The columns represent cubes, and are unique. Each “1” in the KCM represents a way to evaluate the term identified by the subscript.

A valid factorization is a sub-matrix formed from a subset of the rows and columns of the KCM, in which every entry is “1”. Alternatively, a factorization can be considered

$$
\begin{array}{c|ccccccc}
 & a & a^2 & ab & b & b^2 & bc & c \\
\hline
\mathbf{1} & 0 & 1_{(1)} & 1_{(2)} & 0 & 0 & 0 & 1_{(3)} \\
\mathbf{a} & 1_{(1)} & 0 & 0 & 1_{(2)} & 0 & 0 & 0 \\
\mathbf{1} & 0 & 0 & 1_{(4)} & 0 & 1_{(5)} & 1_{(6)} & 0 \\
\mathbf{b} & 1_{(4)} & 0 & 0 & 1_{(5)} & 0 & 0 & 1_{(6)}
\end{array}
$$

Fig. 2: The Kernel Co-Kernel Matrix (KCM) corresponding to the kernels in Equation List 21. Each one in the matrix corresponds to a particular way of evaluating the cube identified by the bracketed subscript.

$$
\begin{array}{c|ccccccc}
 & a & a^2 & ab & b & b^2 & bc & c \\
\hline
\mathbf{1} & 0 & 1_{(1)} & 1_{(2)} & 0 & 0 & 0 & 1_{(3)} \\
\mathbf{a} & \boxed{1_{(1)}} & 0 & 0 & \boxed{1_{(2)}} & 0 & 0 & 0 \\
\mathbf{1} & 0 & 0 & 1_{(4)} & 0 & 1_{(5)} & 1_{(6)} & 0 \\
\mathbf{b} & \boxed{1_{(4)}} & 0 & 0 & \boxed{1_{(5)}} & 0 & 0 & 1_{(6)}
\end{array}
$$

(a) A KCM covering

The rewritten polynomials are:

$$ae + c$$
$$be + bc$$

where:

$$e = a + b$$

(b) The corresponding factorization

Fig. 3: A factorization of our example polynomials from equation list 20 that avoids evaluating $a + b$ twice.

as any block of ones that can be formed by permuting the rows and columns of the KCM. We show a factorization of our polynomials from Equations 20 in Figure 3. The SOP $a + b$ is factored into a new expression, avoiding the repeated evaluation.

The columns of the sub-matrix represent the sum of cubes that form the new sub-term. Each row of the sub-matrix represents a kernel. When the new sub-term is multiplied by the associated co-kernel, it evaluates to one or more terms from the original SOP that produced that kernel.

Factorisations are scored according to a function that describes the number of the multiplications and additions saved:

$$
\begin{aligned}
score =& m * \left((C-1) * \left(R + \sum_{i=0}^{R} M(R_i) \right) + (R-1) * \sum_{j=0}^{C} M(C_j) \right) \\
&+ (R-1) * (C-1)
\end{aligned}
\tag{23}
$$

where:
m is a factor that weights the number of multiplies saved,
R is the number of rows (co-kernels),
C is the number of columns (cubes),
$M(R_i)$ is the number of multiplications required to evaluate co-kernel i,

$M(C_j)$ is the number of multiplications required to evaluate cube j

After the highest scoring factorization from the KCM is found, it is extracted into a new term. The KCM is updated to take account of the terms that no-longer require evaluation and additional rows added for the newly extracted term. This process repeats until no non-zero scoring factorizations are found. This makes the factorization algorithm *greedy*.

Hosangadi et al. do not describe an algorithm to find the best scoring matrix covering in [Hosangadi et al. 2006]. They do however note that it is analogous to the rectangle covering problem described in [Brayton et al. 1987], which is NP-hard. We note that the KCM can effectively be considered as the adjacency matrix of a bipartite graph. Any valid coverings then correspond to bicliques within that graph. We have implemented a branch and bound solver to search for the highest scoring factorizations.

5.2. Extensions to Hosangadi et al. Algorithm

The Hosangadi et al. algorithm treats all literals (numbers or variables) identically. This is problematic since many numerical relationships between expressions are completely opaque to the factorizer.

We consider the following two expressions:

$$e_0 = \frac{3}{5}x + \frac{5}{7}y \tag{24a}$$

$$e_1 = 1\frac{1}{5}x + 1\frac{3}{7}y \tag{24b}$$

The expression e_1 can be calculated by doubling e_0. This requires a single multiply as opposed to the two multiplies and an addition that would be required to evaluate e_1 in isolation. However, when applied to equation set 24, the Hosangadi et al. algorithm will consider the numeric coefficients as four distinct and unrelated values. As a consequence, it is incapable of exploiting the redundancy between the two expressions.

The topological optimizations implemented in FFC [Kirby et al. 2006] demonstrate the importance of exploiting numerical relationships to reduce local assembly operation count. Therefore, we consider it important that we can discern these relationships in our factorizer.

We handle this by decomposing all rational coefficients in our expressions into a product of prime numbers raised to positive and negative exponents. Equation set 24 is rewritten as follows:

$$e_0 = 3^1 5^{-1} x + 5^1 7^{-1} y \tag{25a}$$

$$e_1 = 3^1 2^1 5^{-1} x + 5^1 2^1 7^{-1} y \tag{25b}$$

Given this input, the factorizer can now detect that the expression e_0 is equivalent to $e_1 \div 2$. Our factorizer will produce the following rewritten expressions:

$$e_0 = e_2 \tag{26a}$$

$$e_1 = 2^1 e_2 \tag{26b}$$

$$e_2 = 3^1 5^{-1} x + 5^1 7^{-1} y \tag{26c}$$

This form of decomposition requires that the coefficients in our expressions are represented as rational values. It is for this reason that we avoid introducing any form of numerical approximation during our symbolic manipulation. We can still handle arbitrary floating point values in our expressions if necessary, but these will be opaque to the factorization pass.

Since multiplication and addition of constant values can be performed at compile time or inside our code generator, the functions we use to evaluate the number of multiplies in a cube and the number of operations saved by a factorization need to reflect this.

Within a cube, all multiplies between constants can be evaluated. Therefore, given the cube $c = 3^1 5^{-1} 7^2 x$, the number of multiplies required to evaluate it is $M(c) = 1$. The function used to score factorizations must also be updated. The original Hosangadi et al. factorization scoring function assumes that given the cubes c_1 and c_1, the number of multiplies required to evaluate the expression $c_1 c_2$ is $M(c_1) + M(c_2) + 1$ (i.e. the sum of the multiplies required to evaluate each cube and an additional multiply to compute the product).

We distinguish between different types of cube when computing the original cost to evaluate a product of two cubes. Firstly, we distinguish the *unit cube*, since multiplication by one is always free. Secondly, we observe that multiplication of two cubes c_1 and c_2 only requires $M(c_1) + M(c_2)$ multiplies if both cubes have a numeric coefficient. We identify the three possible cases.

(1) For two compile-time constants, the multiply can be also be done at compile-time. Hence, the cost is $M(c_1) + M(c_2) = 0 + 0 = 0$.
(2) Taking a cube with a numeric coefficient $c_1 = 5ab$ and a constant valued cube $c_2 = 7$, the multiplication between the numeric part of c_1 and c_2 can be performed at compile time. For our example, the cost is $M(c_1) + M(c_2) = 2 + 0 = 2$.
(3) Taking the cubes $c_1 = 5a$ and $c_2 = 7b$, the cost of evaluating the product $35ab$ is $M(c_1) + M(c_2) = 1 + 1 = 2$.

If one or both cubes consist of only values known at run-time, then the cost of evaluating the product of the cubes c_1 and c_2 remains $M(c_1) + M(c_2) + 1$.

We also account for compile-time evaluation of additions, but these occur far less frequently, when at least two terms in a SOP have both a cube and co-kernel that are numeric. This requires that we can also identify when a cube is entirely numeric (as opposed to merely having a numeric coefficient).

6. EVALUATION

We evaluate our approach by comparing against code generated by the FEniCS Form Compiler using both quadrature and tensor contraction implementations for equivalent forms. We enable optimizations for both tensor contraction and quadrature implementations. In the case of tensor contraction, this enables the FErari topological optimizations [Kirby et al. 2006]. In the case of quadrature, this eliminates operations on zeros and performs loop-invariant code motion [Ølgaard and Wells 2010]. We consider two different sets of forms.

The set first consists of the two-dimensional mass matrix, with a Lagrange basis of order q and pre-multiplied by n_f functions of order p. We show an example of the UFL corresponding to this form where $p = 1$, $q = 2$ and $n_f = 3$ in Figure 4. We also show an example of an EXCAFÉ generated `tabulate_tensor` method (as specified by UFC) in Figure 7.

We choose this set because it is also used by Ølgaard and Wells to evaluate the different performance characteristics of tensor contraction and quadrature local assembly implementations ([Ølgaard and Wells 2010], Section 4.3). Ølgaard and Wells state that forms with numerous pre-multiplying functions are typical of the Jacobian resulting

```
element   = FiniteElement("Lagrange", "triangle", 2)
element_f = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)

f = Coefficient(element_f)
g = Coefficient(element_f)
h = Coefficient(element_f)

a = f*g*h*dot(v, u)*dx
```

Fig. 4: A specification for a pre-multiplied mass matrix with element order $q = 2$, pre-multiplied by $n_f = 3$ functions of order $p = 1$.

```
element   = FiniteElement("Lagrange", "triangle", 2)
element_f = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)

f = Coefficient(element_f)

a = f*inner(grad(u), grad(v))*dx
```

Fig. 5: A specification for a pre-multiplied Laplacian with element order $q = 2$, pre-multiplied by $n_f = 1$ function of order $p = 1$.

from the linearisation of non-linear differential equations in a practical simulation. We note that an increased number of pre-multiplying functions leads to an increase in the rank of the geometry tensor in a tensor contraction local assembly implementation.

The second set of forms consist of a two-dimensional Laplacian operator with a Lagrange basis of order q and pre-multiplied by n_f functions of order p. We show an example of the UFL corresponding to this form where $p = 1$, $q = 2$ and $n_f = 3$ in Figure 5.

This choose this set of forms because it allows us to observe the effect of our optimizations in the presence of differential operators. We note that increasing the number of derivatives in a form causes the complexity of tensor contraction based assembly to increase exponentially [Kirby and Logg 2006].

We vary the values of p, q and n_f for both sets of forms to demonstrate the effectiveness of our approach with different form complexities. We compare the number of floating point operations required to evaluate the pre-multiplied mass matrix with and without compiler optimizations enabled in Tables I and II, respectively. We compare the number of operations required to evaluate the pre-multiplied Laplacian operator in Table IV.

Results not present in the table indicate forms that we could not generate code for due to RAM or execution time constraints. We discuss this further in Section 6.3.

Generation of the quadrature and tensor contraction implementations were performed with version 1.0.0 (latest stable version at time of writing) of the FEniCS Form Compiler. The benchmarks were compiled with the GNU C++ Compiler 4.7.1 and the

Intel C++ Compiler 12.0. The benchmarks were run on a 2.4Ghz Intel Core 2 Duo with a 3MB L2 cache running 64-bit Debian "testing".

6.1. Methodology

In order to count the number of floating point operations in each generated local assembly implementation, we have used the Performance Application Programming Interface (PAPI) library [Terpstra et al. 2010]. PAPI allows us to count the number of floating point operations in the compiled local assembly implementations by using processor hardware performance counters. We average our results over at least 100,000 calls to the generated local assembly function.

Since we measure the operation counts of compiled code, these will differ to the theoretical operation count values reported by each code generator and are dependent on compiler optimization effects. However, the results we collect are true reflections of the number of operations required to perform local assembly.

We compile our generated code with both the GNU C++ Compiler 4.7.1 and the Intel C++ Compiler 12.0 to explore the optimization effects of different compilers. Manual inspection of large FFC-generated tensor contraction implementations reveal common sub-expressions that a compiler should be able to take advantage of. Therefore, we expect that the true operation counts may be significantly smaller than code-generator predicted values for those implementations.

Counting the number of operations in compiled code rather than those reported by the respective code generators also gives us an objective scheme that is not dependent on the particular code generator's concept of an operation. This scheme is therefore also unaffected by any flaws in the code generator's calculation of this value.

6.2. Results

Operation counts for EXCAFÉ and optimized FFC-generated local pre-multiplied mass matrix assembly implementations under different compilers (with optimization) are shown in Table I. We also provide FLOP counts for implementations compiled with the GNU C++ Compiler 4.7.1 with optimizations disabled to clarify compiler-dependent effects in Table II.

We observe that operation count results for FFC-generated code vary significantly for different compilers and optimization levels. In particular, the Intel C++ Compiler 12.0 appears to be significantly more effective at reducing the operation count of optimized FFC-generated quadrature and tensor contraction implementations in comparison to the GNU C++ Compiler 4.7.1.

The operation count of EXCAFÉ-generated code varies significantly less than the FFC-generated code under varying compilers and optimization levels. We attribute this to the fact that our factorizer can exploit many of the redundancies utilised by conventional C compilers, in particular those found using common sub-expression elimination. By using a CSE pass targeted towards polynomials, we also aim to exploit redundancies not found by conventional CSE passes.

FErari also attempts to optimize for operation count when generating tensor contraction implementations, but its model prevents it from exploiting certain redundancies that would be detected by compiler common sub-expression elimination passes. We provide a concrete example in Section 7.1.

For the pre-multiplied mass matrix forms we have evaluated, we use fewer flops than at least one of the FFC-generated local assembly implementations. When the number of pre-multiplying functions is greater than one, under the GNU C++ Compiler 4.7.1, we can always reduce operation count over both the tensor contraction and quadrature implementations.

	$n_f = 1$				$n_f = 2$				$n_f = 3$				$n_f = 4$			
	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E
$p = 1, q = 1$	145	28	28	1.00	175	92	73	**1.26**	243	267	122	**1.99**	607	792	215	**2.82**
$p = 1, q = 2$	609	77	91	0.85	1123	217	163	**1.33**	1607	682	280	**2.44**	2682	1831	503	**3.64**
$p = 1, q = 3$	4935	132	161	0.82	7882	511	419	**1.22**	8057	1624	927	**1.75**	11851	2662	1199	**2.22**
$p = 1, q = 4$	17082	455	484	0.94	24847	1178	1065	**1.11**	25099	2854	2148	**1.33**	34503	4356	2874	**1.52**
$p = 2, q = 1$	169	52	55	0.95	583	344	220	**1.56**	1532	2022	919	**1.67**	2671	7361	2426	**1.10**
$p = 2, q = 2$	1111	132	131	**1.01**	2632	1040	591	**1.76**	4255	3999	2354	**1.70**	-	-	-	-
$p = 2, q = 3$	7857	340	352	0.97	11779	2213	1468	**1.51**	16667	6950	4658	**1.49**	-	-	-	-
$p = 2, q = 4$	24811	915	971	0.94	34405	4289	3599	**1.19**	-	-	-	-	-	-	-	-
$p = 3, q = 1$	242	110	92	**1.20**	1607	1020	503	**2.03**	-	-	-	-	-	-	-	-
$p = 3, q = 2$	1607	242	196	**1.23**	4363	2893	1465	**1.97**	-	-	-	-	-	-	-	-
$p = 3, q = 3$	8057	812	808	1.00	16814	6052	4596	**1.32**	-	-	-	-	-	-	-	-
$p = 3, q = 4$	25099	1765	2014	0.88	45959	10364	9683	**1.07**	-	-	-	-	-	-	-	-

(a) GNU C++ Compiler 4.7.1

	$n_f = 1$				$n_f = 2$				$n_f = 3$				$n_f = 4$			
	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E
$p = 1, q = 1$	175	28	31	0.90	175	76	73	**1.04**	213	185	125	**1.48**	530	516	218	**2.37**
$p = 1, q = 2$	476	78	94	0.83	885	176	166	**1.06**	1271	453	287	**1.58**	2133	911	506	**1.80**
$p = 1, q = 3$	2361	123	163	0.75	3832	375	413	0.91	3991	762	935	0.81	5947	1288	1200	**1.07**
$p = 1, q = 4$	7867	408	483	0.84	11541	807	1046	0.77	11758	1268	2157	0.59	16276	1780	2892	0.62
$p = 2, q = 1$	211	51	58	0.88	524	278	224	**1.24**	1457	1365	929	**1.47**	2527	4809	2514	**1.01**
$p = 2, q = 2$	667	135	134	**1.01**	1682	723	604	**1.20**	2851	2404	2438	0.99	-	-	-	-
$p = 2, q = 3$	3837	317	354	0.90	5947	1390	1493	0.93	8689	3495	4715	0.74	-	-	-	-
$p = 2, q = 4$	11557	807	972	0.83	16295	2347	3632	0.65	-	-	-	-	-	-	-	-
$p = 3, q = 1$	266	108	95	**1.14**	1432	781	525	**1.49**	-	-	-	-	-	-	-	-
$p = 3, q = 2$	1223	244	213	**1.15**	3470	1922	1514	**1.27**	-	-	-	-	-	-	-	-
$p = 3, q = 3$	3907	709	834	0.85	8631	3343	4656	0.72	-	-	-	-	-	-	-	-
$p = 3, q = 4$	11636	1516	2021	0.75	21994	5883	9756	0.60	-	-	-	-	-	-	-	-

(b) Intel C++ Compiler 12.0

Table I: The number of floating point operations required to perform local assembly of pre-multiplied mass matrices of varying complexity over a two-dimensional triangular cell. Forms use an order q Lagrangian basis multiplied with n_f functions of order p, also discretized using a Lagrangian basis. Code was compiled using the “-O3” level of optimization. The columns Q, T and E denote the number of floating point operations required by the quadrature, tensor contraction and EXCAFÉ implementations, respectively. The column B/E denotes the improvement in operation count of the EXCAFÉ generated implementation over the quadrature or tensor contraction implementation with the lowest floating point operation count.

For the pre-multiplied Laplacian forms that we have evaluated, we always use fewer operations than both FFC-generated local assembly implementations under the GNU C++ Compiler 4.7.1 and the Intel C++ Compiler 12.0. We provide operation counts for EXCAFÉ and optimized FFC-generated local assembly implementations under different compilers (with optimization) in Table IV.

We observe the reduction in operation count of EXCAFÉ generated code against the tensor and quadrature implementations tend to become greater for increasing values of n_f. We also observe that for fixed values of p and n_f, the greatest reductions in

	$n_f = 1$				$n_f = 2$				$n_f = 3$				$n_f = 4$			
	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E
$p = 1, q = 1$	217	29	34	0.85	259	104	73	**1.42**	350	343	125	**2.74**	679	1034	222	**3.06**
$p = 1, q = 2$	820	87	97	0.90	1484	235	163	**1.44**	2087	867	297	**2.92**	3433	2175	513	**4.24**
$p = 1, q = 3$	4938	144	184	0.78	7894	573	438	**1.31**	8065	1728	934	**1.85**	11859	3070	1201	**2.56**
$p = 1, q = 4$	17097	481	575	0.84	24888	1253	1085	**1.15**	25204	2974	2148	**1.38**	34539	4718	2874	**1.64**
$p = 2, q = 1$	253	60	55	**1.09**	655	396	226	**1.75**	1682	2548	926	**1.82**	2887	11248	2437	**1.18**
$p = 2, q = 2$	1473	151	139	**1.09**	3383	1208	593	**2.04**	5338	4781	2383	**2.01**	-	-	-	-
$p = 2, q = 3$	7863	393	387	**1.02**	11789	2463	1469	**1.68**	16703	7583	4716	**1.61**	-	-	-	-
$p = 2, q = 4$	24892	973	1051	0.93	34509	4451	3599	**1.24**	-	-	-	-	-	-	-	-
$p = 3, q = 1$	350	118	92	**1.28**	1757	1170	503	**2.33**	-	-	-	-	-	-	-	-
$p = 3, q = 2$	2089	300	221	**1.36**	5444	3207	1465	**2.19**	-	-	-	-	-	-	-	-
$p = 3, q = 3$	8067	876	862	**1.02**	16825	6353	4596	**1.38**	-	-	-	-	-	-	-	-
$p = 3, q = 4$	25114	1810	2015	0.90	46089	11122	9752	**1.14**	-	-	-	-	-	-	-	-

Table II: The number of floating point operations required to perform local assembly of pre-multiplied mass matrices of varying complexity over a two-dimensional triangular cell. Code was compiled with the GNU C++ Compiler 4.7.1 at optimization level "-O0". Forms use an order q Lagrangian basis multiplied with n_f functions of order p, also discretized using a Lagrangian basis. The columns Q, T and E denote the number of floating point operations required by the quadrature, tensor contraction and EXCAFÉ implementations, respectively. The column B/E denotes the improvement in operation count of the EXCAFÉ generated implementation over the quadrature or tensor contraction implementation with the lowest floating point operation count.

operation count typically occur at $q = 1$ or $q = 2$ and decrease as q moves away from these values.

Unfortunately, we cannot draw conclusions about the trends we see in reduction across operation count for varying values of p, q and n_f. Since we do not know how close the factorizations found by our algorithm are to the global optimal (the algorithm is greedy), we cannot be certain that the trends we see are not due to properties of the expressions and the factorization algorithm.

We have also evaluated the performance of the generated code by timing the generated assembly loops. We show these results in Table III. We stress that these results only reflect speed-ups on particular architecture, and do not include the costs of fetching cell geometry and field coefficient data from memory or sparse matrix insertion costs.

Our results suggest that the reduction in operation count typically corresponds with an increase in performance on the architecture on which we evaluated. Since the code we generate is effectively an unrolled implementation (we do not generate any loops) the increased code size might become a factor for more complex forms. However, we do not appear to see this for our chosen benchmarks.

6.3. Scalability

Currently, we cannot scale our code generation to some of the more complex pre-multiplied mass matrix examples. This is a direct consequence of the complexity of expressions generated, which can be computationally expensive to symbolically integrate, and both computationally intensive and memory consuming to factorize.

As discussed in Section 4.4, we use a scheme for symbolic integration that involves selective expansion of the input expression before performing the integration itself. For our benchmarks, we find that this strategy scales well memory-wise, causing time to become the more limiting factor.

	$n_f = 1$				$n_f = 2$				$n_f = 3$				$n_f = 4$			
	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E
$p = 1, q = 1$	77	19	18	**1.06**	94	46	29	**1.59**	123	132	41	**2.96**	272	494	83	**3.29**
$p = 1, q = 2$	423	43	43	1.00	709	111	94	**1.18**	989	384	112	**3.42**	1578	1104	223	**4.94**
$p = 1, q = 3$	1758	96	100	0.95	2773	258	188	**1.37**	2893	832	446	**1.87**	4211	1714	643	**2.67**
$p = 1, q = 4$	6949	273	345	0.79	11419	831	615	**1.35**	10406	3278	1450	**2.26**	15377	7217	3282	**2.20**
$p = 2, q = 1$	99	26	23	**1.13**	240	169	95	**1.77**	564	1351	461	**1.22**	1290	13971	1675	0.77
$p = 2, q = 2$	1089	80	81	0.98	2014	781	370	**2.11**	2975	7058	1698	**1.75**	-	-	-	-
$p = 2, q = 3$	4280	251	267	0.94	6219	1956	1251	**1.56**	8762	13569	9939	0.88	-	-	-	-
$p = 2, q = 4$	12940	701	770	0.91	16444	7901	7306	**1.08**	-	-	-	-	-	-	-	-
$p = 3, q = 1$	196	72	65	**1.12**	901	964	410	**2.20**	-	-	-	-	-	-	-	-
$p = 3, q = 2$	1551	149	141	**1.05**	2622	2207	911	**2.42**	-	-	-	-	-	-	-	-
$p = 3, q = 3$	3976	660	584	**1.13**	8662	11858	9881	0.88	-	-	-	-	-	-	-	-
$p = 3, q = 4$	13199	1418	1790	0.79	23628	21189	21462	0.99	-	-	-	-	-	-	-	-

(a) GNU C++ Compiler 4.7.1

	$n_f = 1$				$n_f = 2$				$n_f = 3$				$n_f = 4$			
	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E
$p = 1, q = 1$	108	20	23	0.85	93	38	29	**1.31**	120	83	43	**1.93**	242	232	75	**3.11**
$p = 1, q = 2$	470	48	50	0.96	873	70	63	**1.11**	1183	172	107	**1.60**	1864	387	204	**1.89**
$p = 1, q = 3$	2161	95	110	0.86	3193	158	191	0.83	3310	315	389	0.81	4724	546	501	**1.09**
$p = 1, q = 4$	6289	235	334	0.70	9309	348	485	0.72	9441	551	1055	0.52	15096	923	2063	0.45
$p = 2, q = 1$	139	25	26	0.94	268	116	83	**1.39**	689	650	341	**1.91**	1411	9273	1168	**1.21**
$p = 2, q = 2$	586	68	73	0.93	1073	304	220	**1.38**	1656	1124	965	**1.16**	-	-	-	-
$p = 2, q = 3$	4550	219	242	0.90	7130	883	941	0.94	9254	7420	8913	0.83	-	-	-	-
$p = 2, q = 4$	14168	507	749	0.68	17928	1520	7046	0.22	-	-	-	-	-	-	-	-
$p = 3, q = 1$	275	68	54	**1.26**	837	417	212	**1.97**	-	-	-	-	-	-	-	-
$p = 3, q = 2$	1569	114	118	0.96	4566	1490	1032	**1.44**	-	-	-	-	-	-	-	-
$p = 3, q = 3$	5118	462	541	0.85	10539	8199	9651	0.85	-	-	-	-	-	-	-	-
$p = 3, q = 4$	14033	864	1533	0.56	26110	14052	19999	0.70	-	-	-	-	-	-	-	-

(b) Intel C++ Compiler 12.0

Table III: The times (nanoseconds) required to execute local assembly of pre-multiplied mass matrices of varying complexity over a two-dimensional triangular cell. Forms use an order q Lagrangian basis multiplied with n_f functions of order p, also discretized using a Lagrangian basis. The columns $\boldsymbol{Q}$, $\boldsymbol{T}$ and $\boldsymbol{E}$ denote the execution times of the quadrature, tensor contraction and EXCAFÉ implementations, respectively. The column $\boldsymbol{B/E}$ denotes the improvement in execution time of the EXCAFÉ generated implementation over the quadrature or tensor contraction implementation with the lowest execution time.

For complex forms, the kernel cube matrix we construct for factorization can require significant memory to store and time to search. We note that the algorithm we use for selecting expressions to factorize is optimal (although it is applied repeatedly in a greedy manner). We believe it is possible to avoid storing this matrix explicitly, reducing memory costs at the expense of time. However, a heuristic approach to locating the most beneficial factorizations may significantly reduce the cost of searching. Since we have been investigating the maximum reduction in operation count our techniques can provide, we have yet to investigate such heuristics.

We note that for the two-dimensional pre-multiplied mass matrix example, the number of terms in expressions we apply factorization to is $O(p^{2n_f})$. Hence, our expression

	$n_f = 1$				$n_f = 2$			
	Q	T	E	B/E	Q	T	E	B/E
$p = 1, q = 1$	49	74	45	**1.09**	128	199	58	**2.21**
$p = 1, q = 2$	1897	289	246	**1.17**	1939	854	428	**2.00**
$p = 1, q = 3$	6974	953	798	**1.19**	12097	3145	1601	**1.96**
$p = 1, q = 4$	38061	2405	1842	**1.31**	59691	7787	3684	**2.11**
$p = 2, q = 1$	125	75	45	**1.67**	305	497	100	**3.05**
$p = 2, q = 2$	1943	560	446	**1.26**	4012	3087	1069	**2.89**
$p = 2, q = 3$	11999	1956	1428	**1.37**	25338	12311	4425	**2.78**
$p = 2, q = 4$	59689	5173	3563	**1.45**	86408	33112	11840	**2.80**
$p = 3, q = 1$	275	208	58	**3.59**	923	1777	195	**4.73**
$p = 3, q = 2$	2312	1049	658	**1.59**	8724	10186	2691	**3.24**
$p = 3, q = 3$	16162	3214	2124	**1.51**	37038	36775	11204	**3.28**
$p = 3, q = 4$	59929	9360	5767	**1.62**	-	-	-	-

(a) GNU C++ Compiler 4.7.1

	$n_f = 1$				$n_f = 2$			
	Q	T	E	B/E	Q	T	E	B/E
$p = 1, q = 1$	54	76	48	**1.12**	136	199	61	**2.23**
$p = 1, q = 2$	1201	295	251	**1.18**	1249	653	443	**1.47**
$p = 1, q = 3$	4953	916	802	**1.14**	8482	2168	1611	**1.35**
$p = 1, q = 4$	26192	2106	1837	**1.15**	41141	5021	3679	**1.36**
$p = 2, q = 1$	178	85	48	**1.77**	317	481	103	**3.08**
$p = 2, q = 2$	1237	538	464	**1.16**	2617	2237	1093	**2.05**
$p = 2, q = 3$	8477	1795	1447	**1.24**	17984	7884	4448	**1.77**
$p = 2, q = 4$	41067	4457	3544	**1.26**	59626	19174	11903	**1.61**
$p = 3, q = 1$	349	208	61	**3.41**	937	1566	198	**4.73**
$p = 3, q = 2$	1460	967	664	**1.46**	5700	7079	2724	**2.09**
$p = 3, q = 3$	11415	2941	2164	**1.36**	26255	23605	11246	**2.10**
$p = 3, q = 4$	41158	7631	5782	**1.32**	-	-	-	-

(b) Intel C++ Compiler 12.0

Table IV: The number of floating point operations required to perform local assembly of pre-multiplied Laplacian matrices of varying complexity over a two-dimensional triangular cell. Forms use an order q Lagrangian basis multiplied with n_f functions of order p, also discretized using a Lagrangian basis. Code was compiled using the "-O3" level of optimization. The columns $\boldsymbol{Q}$, $\boldsymbol{T}$ and $\boldsymbol{E}$ denote the number of floating point operations required by the quadrature, tensor contraction and EXCAFÉ implementations, respectively. The column $\boldsymbol{B/E}$ denotes the improvement in operation count of the EXCAFÉ generated implementation over the quadrature or tensor contraction implementation with the lowest floating point operation count.

sizes increase exponentially as the number of multiplying functions increases, and also as a square of the rank of the pre-multiplying basis.

For affine mappings, q does not affect the complexity of the individual expressions we factorize since the symbolic integration step transforms the cell basis function expressions to constant values. However, as q increases, so does the size of the local assembly matrix, so the factorizer must handle greater numbers of expressions.

We give an indication of the time and memory requirements required to compile some of the pre-multiplied mass matrix forms in Table V.

	$n_f = 1$				$n_f = 2$				$n_f = 3$				$n_f = 4$			
	Seconds		MB		Seconds		MB		Seconds		MB		Seconds		MB	
	T	E	T	E	T	E	T	E	T	E	T	E	T	E	T	E
$p = 1, q = 1$	0	1	20	32	0	1	20	32	0	1	20	32	1	7	20	37
$p = 1, q = 2$	1	2	21	32	1	3	21	32	2	6	21	33	4	27	21	59
$p = 1, q = 3$	4	4	22	32	6	8	22	32	11	36	22	46	23	201	22	149
$p = 1, q = 4$	18	13	27	32	27	28	27	37	52	142	27	92	121	922	27	502
$p = 2, q = 1$	1	2	20	48	1	3	21	47	2	52	21	77	6	3725	22	598
$p = 2, q = 2$	2	5	20	47	4	15	21	47	15	563	21	155	-	-	-	-
$p = 2, q = 3$	3	5	22	47	9	63	22	69	43	4286	23	1303	-	-	-	-
$p = 2, q = 4$	23	22	27	47	67	636	27	214	-	-	-	-	-	-	-	-
$p = 3, q = 1$	1	2	21	66	1	32	21	71	-	-	-	-	-	-	-	-
$p = 3, q = 2$	1	5	21	66	5	601	21	156	-	-	-	-	-	-	-	-
$p = 3, q = 3$	6	19	22	66	32	9188	22	1074	-	-	-	-	-	-	-	-
$p = 3, q = 4$	33	61	27	67	167	8268	27	693	-	-	-	-	-	-	-	-

Table V: The user time and maximum resident process sizes of FFC performing tensor contraction code generation (T) and EXCAFÉ code generation (E). We show results for the pre-multiplied mass matrix examples across a range of problem sizes. Information was collected using the GNU "time" command.

In comparison to FFC (when generating optimized tensor contraction code), we often have significantly greater time and memory requirements. We attribute this to the difference in size and nature of the search spaces traversed in order to locate beneficial optimizations.

The FErari topological optimizations employ a minimal spanning tree, for which there exist efficient calculation techniques. We also note that the optimizations found by this technique do not require the introduction of new sub-expressions. However, we have found that certain numerical redundancies can only be exploited though sub-expression introduction (Section 7.1).

Furthermore, the FErari topological optimization techniques are optimal with respect to the reduction of operation count that can be expressed using those techniques. The fact that our (and other [Kirby and Scott 2007]) techniques can improve upon them also suggests that the topological optimizations operate within a more restrictive search space.

The techniques we have adapted from Hosangadi et al. [Hosangadi et al. 2006] require the solution of a covering problem which is NP-Hard [Brayton et al. 1987]. Our technique of decomposing rational coefficients increases the size of this search space further.

Any redundancy found via a complexity-reducing relation can also be exploited by our CSE pass. In addition, we have the capacity to exploit many redundancies that cannot be expressed through complexity reducing relations. However, the algorithmic complexity of locating solutions that effectively exploit these redundancies is significantly greater.

6.4. Validation and Numerical Accuracy

Given the extensive amount of expression manipulation performed by EXCAFÉ to produce the generated code, we considered it important to validate our results. We specifically implemented our cell entity numbering and basis function construction so that given the same input data, our generated code should produce the same output as the FFC generated implementations.

For each tested form, we generated pseudo-random basis function coefficients between -1 and 1 and provided them as input to the FFC generated quadrature implementation and EXCAFÉ generated code. We used pseudo-randomly generated locations on a circle for the cell vertex locations. We verified that the Frobenius norm of the differences between the two matrices remained less than 10^{-10}.

During our validation, we observed significant differences between the local assembly matrix entries computed by the FFC-generated tensor contraction implementations and both the EXCAFÉ-generated implementations and the FFC-generated quadrature implementations. Disabling the tensor contraction topological optimizations caused these differences to become negligible. We have reported our observations to the FEniCS developers[2].

For the randomly generated basis function coefficients and cell geometry already described, we also measured the extent of these differences across our class of premultiplied mass matrix variational forms. The Frobenius norms of the differences between the FFC-generated quadrature implementation and the EXCAFÉ and optimized tensor contraction implementations are shown in Table VI.

We observe that for more complex forms, the accuracy of the optimized tensor contraction implementations is significantly reduced. By editing the FErari source to disable collinearity optimizations, we determined that the Hamming distance optimizations were the cause of the inaccuracies.

We rewrote the `tabulate_tensor` function (which performs local assembly) of a FFC-generated optimized tensor contraction to use GCC's non-standard 128-bit floating point type (`__float128`). Since we saw the same numerical errors, we determined that the precision loss we saw was not occurring at run-time, but instead within the code generator.

Within FErari's topological optimization code, we found two approximations that contributed to the numerical error:

(1) The Hamming distance optimization generates code that derives an inner product $a_1 \cdot g_0$ from $a_0 \cdot g_0$ (where the entries of a_0 and a_1 are known at code-generation time). It computes the value $(a_1 - a_0) \cdot g_0$ and can avoid generating multiplications where entries of $a_1 - a_0$ are known to be zero.
Since FFC uses floating point, it approximates the check to see if two coefficients are equal by checking to see if they differ by less than a chosen ϵ value.

(2) When unrolling code to compute an inner product $a_0 \cdot g_0$ from scratch, FFC will neglect to compute terms where the coefficient in a_0 is less than some ϵ. This approximates a check for coefficients in a_0 that should be zero due to term cancellation but are non-zero due to floating point precision issues.

We observe that as forms become more complex, the inner product coefficients tend to become smaller. The numerical inaccuracies we see occur when the coefficients become smaller than the chosen ϵ values, causing legitimate terms to not be evaluated in unrolled inner products. Changing the ϵ values to zero causes the numerical inaccuracies to disappear, but has the effect of eliminating the reduction in operation count from the topological optimizations. We note that for the FFC-generated tensor contraction implementations that have accuracy issues, we expect a sufficiently accurate implementation to require more floating point operations than the results currently presented in Table I.

One strategy to solve this would be to use an adaptive scheme that chooses ϵ values based on the coefficient values in the reference tensor. However, this scheme is still problematic in the sense that it is impossible to determine with complete certainty

[2]`https://bugs.launchpad.net/ferari/+bug/822775`

	$n_f = 1$		$n_f = 2$		$n_f = 3$		$n_f = 4$	
	EXCAFÉ	Tensor	EXCAFÉ	Tensor	EXCAFÉ	Tensor	EXCAFÉ	Tensor
$p = 1, q = 1$	$6.89e{-}17$	$9.30e{-}17$	$5.49e{-}17$	$4.95e{-}17$	$1.44e{-}17$	$1.34e{-}17$	$1.56e{-}17$	$\mathbf{4.43e{-}5}$
$p = 1, q = 2$	$6.80e{-}17$	$1.56e{-}16$	$1.22e{-}16$	$7.14e{-}17$	$1.17e{-}17$	$1.27e{-}17$	$8.05e{-}18$	$\mathbf{9.77e{-}5}$
$p = 1, q = 3$	$8.16e{-}17$	$8.94e{-}17$	$6.21e{-}17$	$\mathbf{6.77e{-}5}$	$2.44e{-}17$	$\mathbf{7.69e{-}4}$	$8.61e{-}17$	$\mathbf{5.02e{-}4}$
$p = 1, q = 4$	$3.70e{-}16$	$\mathbf{1.83e{-}4}$	$7.12e{-}16$	$\mathbf{4.11e{-}4}$	$2.65e{-}16$	$\mathbf{5.42e{-}4}$	$5.03e{-}16$	$\mathbf{4.36e{-}4}$
$p = 2, q = 1$	$2.42e{-}16$	$2.34e{-}16$	$1.18e{-}16$	$3.42e{-}17$	$2.78e{-}17$	$\mathbf{1.30e{-}4}$	$1.22e{-}16$	$\mathbf{3.06e{-}4}$
$p = 2, q = 2$	$4.50e{-}16$	$1.47e{-}16$	$7.47e{-}17$	$\mathbf{7.04e{-}5}$	$4.05e{-}16$	$\mathbf{6.47e{-}4}$	-	-
$p = 2, q = 3$	$1.85e{-}16$	$\mathbf{1.65e{-}4}$	$7.76e{-}16$	$\mathbf{5.57e{-}4}$	$1.88e{-}15$	$\mathbf{1.47e{-}3}$	-	-
$p = 2, q = 4$	$2.22e{-}15$	$\mathbf{2.90e{-}4}$	$4.01e{-}15$	$\mathbf{1.42e{-}3}$	-	-	-	-
$p = 3, q = 1$	$6.54e{-}17$	$5.96e{-}17$	$8.13e{-}17$	$\mathbf{9.70e{-}5}$	-	-	-	-
$p = 3, q = 2$	$9.43e{-}17$	$1.26e{-}16$	$8.99e{-}16$	$\mathbf{5.23e{-}4}$	-	-	-	-
$p = 3, q = 3$	$2.27e{-}16$	$\mathbf{2.86e{-}4}$	$5.77e{-}15$	$\mathbf{1.81e{-}3}$	-	-	-	-
$p = 3, q = 4$	$2.45e{-}15$	$\mathbf{5.31e{-}4}$	$1.62e{-}14$	$\mathbf{2.54e{-}3}$	-	-	-	-

Table VI: Frobenius norms of local assembly matrix differences between a reference FFC-generated quadrature implementation and EXCAFÉ and optimized FFC-generated tensor contraction implementations. We have highlighted differences greater than 10^{-10} which suggest that the optimized tensor contraction implementation has computed an incorrect result.

whether coefficients are meant to be equivalent or merely appear to be so within some degree of error.

The symbolic approach used by EXCAFÉ has none of these issues. Since we use rational values throughout our symbolic manipulation, we can always determine correctly whether coefficients or terms are identical, an exact multiple of each other and whether they cancel. Additionally, modifying FFC to use symbolic techniques to compute the value of the reference tensor would also enable the FErari topological analyses to be applied with no need for approximate numerical comparisons.

7. COMPARISON AGAINST GRAPH-BASED OPTIMIZATIONS

We primarily compare EXCAFÉ-generated code to the optimized tensor contraction code generated by FFC using the FErari topological optimizations since we have access to code generated by both. We compare against other graph-based optimizations where appropriate.

We present extracts of FFC-generated optimized tensor contraction code and EXCAFÉ generated code for the same problem in Figures 6 and 7, respectively.

7.1. Sub-expression Introduction

The most apparent difference between EXCAFÉ-generated code and the optimized tensor contraction code is that the EXCAFÉ-generated code introduces new sub-expressions whilst the FErari optimized code does not. This directly affects the types of numerical redundancies we are able to exploit.

The graph-based optimizations generate optimized code by deriving elements of the local assembly tensor from one or more previously computed elements when it will save operations. We consider a case when redundancies can only be exploited by introducing a new sub-expression. Given two elements of the local assembly matrix m_0 and m_1:

$$m_0 = rg_0 + sg_1 + tg_2 + ug_3 + vg_4 + wg_5 \quad (27a)$$

$$m_1 = rg_0 + sg_1 + tg_2 + xg_6 + yg_7 + zg_8 \quad (27b)$$

```
void tabulate_tensor(double* A, const double * const * w,
                     const ufc::cell& c) const
{
  // Extract vertex coordinates
  const double * const * x = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = x[1][0] - x[0][0];
  const double J_01 = x[2][0] - x[0][0];
  const double J_10 = x[1][1] - x[0][1];
  const double J_11 = x[2][1] - x[0][1];

  // Compute determinant of Jacobian
  double detJ = J_00*J_11 - J_01*J_10;

  // Set scale factor
  const double det = std::abs(detJ);

  // Compute geometry tensor
  const double G0_0 = det*w[0][0]*(1.0);
  const double G0_1 = det*w[0][1]*(1.0);
  const double G0_2 = det*w[0][2]*(1.0);

  // Compute element tensor
  A[1] = 0.0166666666666667*G0_0 + 0.0166666666666667*G0_1 + 0.00833333333333332*G0_2;
  A[5] = A[1] - 0.00833333333333334*G0_0 + 0.00833333333333334*G0_2;
  A[0] = A[5] + 0.0416666666666666*G0_0;
  A[7] = A[5];
  A[8] = A[1] + 0.0416666666666666*G0_2;
  A[3] = A[1];
  A[4] = A[1] + 0.0333333333333333*G0_1 + 0.00833333333333334*G0_2;
  A[6] = A[4] - 0.0416666666666666*G0_1;
  A[2] = A[6];
}
```

Fig. 6: An FFC-generated tensor contraction local assembly implementation for a two-dimensional pre-multiplied mass matrix with $p = 1$, $q = 1$ and $n_f = 1$. Some comments were removed and code reformatted for the purposes of clarity.

The variables $r, s, t, u, v, w, x, y, z$ represent elements of the reference tensor and are present in both m_0 and m_1 when the corresponding elements of the reference tensor are identical.

As written above, m_0 and m_1 together take 10 additions and 12 multiplies to evaluate. The presence of identical corresponding elements of the reference tensor make it possible to apply the Hamming distance optimization:

$$m_0 = rg_0 + sg_1 + tg_2 + ug_3 + vg_4 + wg_5 \tag{28a}$$

$$m_1 = m_0 + -ug_3 + -vg_4 + -wg_5 + xg_6 + yg_7 + zg_8 \tag{28b}$$

Now, m_0 and m_1 together take 11 additions and 13 multiplies to evaluate. Since both m_0 and m_1 now incorporate the sum $ug_3 + vg_4 + wg_5$ (m_1 uses it negated), we would expect a reasonably sophisticated compiler reduce the operation count below that. However, this is a compiler-dependent optimization and not apparent to FErari. Since the rewritten m_0 and m_1 require more operations to evaluate than the original, the FErari

```
void tabulate_tensor(double* const A, const double* const* w,
                     const ufc::cell& c) const
{
  const double * const * x = c.coordinates;

  const double var_0 = -x[0][1];
  const double var_1 = x[1][1] + var_0;
  const double var_2 = -x[0][0];
  const double var_3 = x[2][0] + var_2;
  const double var_4 = x[1][0] + var_2;
  const double var_5 = x[2][1] + var_0;
  const double var_6 = -var_1*var_3 + var_4*var_5;
  const double var_7 = std::abs(var_6);
  const double var_8 = 0.0166666666666666664353702*var_7*w[0][0];
  const double var_9 = 0.0166666666666666664353702*var_7*w[0][1];
  const double var_10 = var_8 + var_9;
  A[8] = 0.0500000000000000027755576*var_7*w[0][2] + var_10;
  const double var_11 = 0.0166666666666666664353702*var_7*w[0][2];
  const double var_12 = var_9 + var_11;
  A[5] = 0.00833333333333333332176851*var_7*w[0][0] + var_12;
  A[7] = A[5];
  const double var_13 = var_8 + var_11;
  A[2] = 0.00833333333333333332176851*var_7*w[0][1] + var_13;
  A[1] = 0.00833333333333333332176851*var_7*w[0][2] + var_10;
  A[6] = A[2];
  A[3] = A[1];
  A[0] = 0.0500000000000000027755576*var_7*w[0][0] + var_12;
  A[4] = 0.0500000000000000027755576*var_7*w[0][1] + var_13;
}
```

Fig. 7: An EXCAFÉ generated local assembly implementation for a two-dimensional pre-multiplied mass matrix with $p = 1$, $q = 1$ and $n_f = 1$. We note that the KCM is only intended to enable finding sums of at least two terms to be factored into new expressions. Hosangadi et al. describe an additional algorithm for extracting products common to individual cubes. This would enable products such as `var[7]*w[0][1]` to be moved into new expressions. However, we have found that the common sub-expression elimination in the GCC and ICC compilers can perform these optimizations effectively and so have not yet chosen to implement this pass.

topological optimizations will choose not to exploit any relationship between m_0 and m_1.

Since EXCAFÉ has the ability to introduce new sub-expressions, it can potentially exploit the redundancy between m_0 and m_1 by introducing the new shared expression e:

$$e = rg_0 + sg_1 + tg_2 \tag{29a}$$
$$m_0 = e + ug_3 + vg_4 + wg_5 \tag{29b}$$
$$m_1 = e + xg_6 + yg_7 + zg_8 \tag{29c}$$

Rewritten this way, m_0 and m_1 take 8 additions and 9 multiplies to evaluate, and there are no further redundancies that a compiler could exploit.

Our example suggests that when optimizing for operation count, it is necessary to take account of the introduction of common sub-expressions since many numerical redundancies can only be exploited in this manner. Table I shows that the FLOP count of

EXCAFÉ-generated code remains more consistent across compiler than FFC-generated tensor contraction code, suggesting that we detect exploitable redundancies more effectively.

7.2. Exploitable Redundancies

The Hosangadi et al. CSE pass is capable of moving a sum common to multiple polynomial expressions into a new sub-expression. Hence, we can detect equivalent redundancies to the Hamming distance optimizations used by FErari. As described in Section 7.1, we may be able to reduce operation count further than FFC for the same redundancies.

The Hosangadi et al. CSE algorithm decomposes each sum of products into smaller sums that when multiplied by a specific product, form a subset of terms from the original sum. As described in Section 5.2, we represent our numeric coefficients as rational numbers, and present them to the CSE algorithm as products of primes raised to positive and negative exponents.

As a consequence, our factorization algorithm can identify collinear expressions by removing common factors from one or both expressions. Since we can do this for subsets of terms in a sum, we should also be able to detect the same redundancies found by the partial collinearity optimization described by Wolf and Heath [Wolf and Heath 2009].

Wolf and Heath describe how coplanarity between vectors in the reference tensor enable the calculation of an element of the local assembly matrix from a linear combination of two other elements. Kirby and Scott generalise this optimization using linear dependence between vectors in the reference tensor to determine when an element of the local assembly matrix can be computed from a weighted sum of other elements of the local assembly matrix.

The representation EXCAFÉ uses for common sub-expression elimination cannot represent the evaluation of a product of variables as a sum. Therefore, it cannot exploit coplanarity and linear dependence redundancies in the general case. However, if the set of linearly dependent vectors do not have non-zero elements at identical locations, EXCAFÉ can still exploit these relationships as individual products are not constructed using sums. For example, given the linearly dependent expressions:

$$m_0 = 7g_0 + 5g_2 + 3g_3 \tag{30a}$$
$$m_1 = 6g_1 + 12g_4 + 4g_5 \tag{30b}$$
$$m_2 = 7g_0 + 3g_1 + 5g_2 + 3g_3 + 6g_4 + 2g_5 \tag{30c}$$

EXCAFÉ will rewrite them as follows:

$$e_0 = 7g_0 + 5g_2 + 3g_3 \tag{31a}$$
$$e_1 = 3g_1 + 6g_4 + 2g_5 \tag{31b}$$
$$m_0 = e_0 \tag{31c}$$
$$m_1 = 2e_1 \tag{31d}$$
$$m_2 = e_0 + e_1 \tag{31e}$$

The Hosangadi et al. CSE algorithm chooses to create new sub-expressions based on the number of operations that would be saved compared to a naïve evaluation of the factorized terms. This process is inherently iterative, and repeats until no more operation count reducing factorizations can be found. Hence, the algorithm is greedy (although we choose the best factorization at each step).

We do not yet have a way to optimize for operation count in a more global sense (such as the minimal spanning tree used by Kirby et al. [Kirby et al. 2006]). Our CSE pass is capable of generating a superset of the factorizations exploited by FFC-generated optimized tensor contraction implementations. We believe that this greedy nature of out algorithm accounts for the instances where we generate code that uses more operations that the FFC-generated implementations.

8. CONCLUSION

We have presented the algorithms underpinning a library we implemented, EXCAFÉ, that generates finite element local assembly implementations from specifications of bilinear forms. Symbolic techniques are used for manipulating basis functions and performing symbolic integration.

We have taken existing work by Hosangadi et al. and used it to search for factorizations in local assembly matrix expressions that can take advantage of distributivity, which has typically not been considered in previous work. We have extended the algorithm to make it aware of the multiplicative relationships between numeric coefficients, enabling it to exploit a new class of common sub-expressions for reducing operation count.

We have shown that these techniques can be used to produce local assembly implementations that reduce operation count compared to both quadrature and tensor contraction implementations, in some cases, by over a factor of 4 in compiled code. We also show performance results that indicate that (neglecting other run-time costs) that these reductions in operation count correspond to observable speed-ups against quadrature and optimized tensor implementations.

By using symbolic techniques, our code generator is capable of representing numerical coefficients as rational values throughout its manipulation of expressions. This enables us to avoid committing to floating point values (and possibly losing accuracy) until the step of code generation.

Scalability of these techniques is still an issue. However, the main motivation of this work is to investigate their effectiveness in comparison to other code generation techniques. We believe that with additional work, scalability of both the symbolic integration and factorization algorithms can be improved further.

The symbolic techniques we employ enable us to determine equivalence between and relationships between terms and coefficients in the expressions we optimize with complete accuracy. We have seen that using floating point representation of coefficients can lead to significant numerical issues in form compilers that attempt to infer these relationships. We conclude that the symbolic approach offers significant benefits for any analysis that requires establishing relationships between numeric values.

The code we generate is usually competitive in terms of operation count against the better performing FFC-generated local assembly implementation and always uses fewer operations than the worse-performing implementation. We note that for forms where the optimized tensor contraction implementation has lost accuracy, our reduction in operation count is likely an underestimation since the FFC-generated code has neglected to perform certain numerical operations. As a consequence, we expect that our results will improve against a version of FFC that fixes the topological optimization issues.

The tensor contraction implementations we have compared against used *complexity-reducing relations* to reduce operation count. Related work has used linear dependence relationships in the reference tensor to reduce the operation count of tensor contraction [Kirby and Scott 2007]. Other work has generalised the techniques present in FFC further to exploit relationships such as partial collinearity [Wolf and Heath 2009]. These techniques are able to improve upon the results produced by complexity-reducing relations; comparisons are an interesting direction for future research and we discuss these techniques further in Section 9. We note that both the linear dependence techniques and extensions to complexity-reducing relations do not require the introduction of new sub-expressions, which we identified in Section 7.1 as necessary for exploiting some numerical relationships.

9. FUTURE WORK

Our most interesting results occur at the limits at which we can currently scale our code generation techniques. Further work to extend this frontier will help characterise the performance trends of our techniques at the extremes of our parameter space.

Similarly, we would like to be able to compare against FErari topologically optimized code where the reference tensor has been computed symbolically. Currently, the FErari optimized code produces inaccurate values for more complex forms, making performance comparisons less illustrative. Computing the reference tensor symbolically would enable us to compare operation counts for more complex forms with the knowledge that both codes are evaluating identical expressions.

Related work has explored generalizing complexity-reducing relationships beyond what is currently used by FErari as well as exploiting numerical relationships based on linear dependence [Kirby and Scott 2007; Wolf and Heath 2009]. These techniques have been shown to reduce the operation count over complexity-reducing relations in various instances. As with our techniques, scalability becomes an issue and heuristics are necessary for acceptable performance.

We do not yet have performance results for these techniques and are therefore interested in comparing the limits to which operation count can be reduced by each, and which parameters affect whether a given problem can be solved tractably. We have attempted to expose a large search space for possible optimizations though our CSE approach. A comparison against other techniques should help us to gain further insight into what optimizations we can and cannot expose.

We are also interested in extending our techniques to exploit linear dependence. To fully exploit linear dependence, we would need to be able to detect partial linear dependence, a generalisation of the partial collinearity optimization. However, it seems unlikely that it is possible to do this efficiently, even for small problems. Attempts to combine complexity-reducing relations with limited forms of linear dependence have encountered significant scalability issues [Kirby and Scott 2007] and our proposed search space is much larger than this.

We have restricted our code generation to affine mappings since these are most common and our primary basis for comparison (FFC) does not support non-affine mappings at present. Our primary obstacle in handing non-affine mappings is the symbolic integration step that we perform on the expressions of the local assembly matrix. We currently use our own implementation of symbolic integration optimized for (and only applicable to) polynomials. To scale effectively, we suspect interfacing with a computer algebra system such as MAXIMA (an open-source fork of MACSYMA [Fateman 1989]) would be necessary. Our factorizer is only presented with the polynomial parts of expressions so the non-affine mappings would not cause an issue for this step.

We have yet to systematically determine which redundancies our code generation system makes use of when it reduces operation count over other schemes. To do this would require automated analysis of our generated code since our best results occur on more complex problems. This could lead to a more structured, targeted optimization pass that operates more efficiently than our presented techniques.

So far, we have chosen to optimize only for operation count. However, each new sub-expression we introduce requires additional memory to store. An alternative code generation scheme might attempt to restrict the number of temporary values required while minimizing the number of additional operations. This would enable more effective utilization of hardware on architectures with limited high-bandwidth memory.

REFERENCES

ALNÆS, M. S., LOGG, A., MARDAL, K.-A., SKAVHAUG, O., AND LANGTANGEN, H. P. 2009. Unified framework for finite element assembly. *International Journal of Computational Science and Engineering 4,* 4, 231–244.

ALNÆS, M. S. AND MARDAL, K.-A. 2010. On the efficiency of symbolic computations combined with code generation for finite element methods. *ACM Trans. Math. Softw. 37*, 6:1–6:26.

BANGERTH, W., HARTMANN, R., AND KANSCHAT, G. 2007. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw. 33,* 4, 24:1–24:27.

BAUER, C., FRINK, A., AND KRECKEL, R. 2002. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *Journal of Symbolic Computation 33,* 1, 1–12.

BRAYTON, R. K., RUDELL, R., SANGIOVANNI-VINCENTELLI, A., AND WANG, A. R. 1987. Multi-level logic optimization and the rectangular covering problem. In *International Conference on Computer Aided Design*. 66–69.

BRUASET, A. M. AND LANGTANGEN, H. P. 1997. A comprehensive set of tools for solving partial differential equations; Diffpack. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dhlen and A. Tveito, Eds. Birkhäuser, 61–90.

DUBINER, M. 1991. Spectral methods on triangles and other domains. *J. Sci. Comput. 6*, 345–390.

FATEMAN, R. J. 1989. A review of Macsyma. *IEEE Trans. on Knowl. and Data Eng. 1*, 133–145.

HARRINGTON, S. J. 1979. A new symbolic integration system in reduce. *The Computer Journal 22,* 2, 127–131.

HOSANGADI, A., FALLAH, F., AND KASTNER, R. 2006. Optimizing polynomial expressions by algebraic factorization and common subexpression elimination. *IEEE Trans. on CAD of Integrated Circuits and Systems 25,* 10, 2012–2022.

KIRBY, R. C. 2004. Algorithm 839: FIAT, a new paradigm for computing finite element basis functions. *ACM Trans. Math. Softw. 30*, 502–516.

KIRBY, R. C. AND LOGG, A. 2006. A compiler for variational forms. *ACM Trans. Math. Softw. 32,* 3, 417–444.

KIRBY, R. C., LOGG, A., SCOTT, L. R., AND TERREL, A. R. 2006. Topological optimization of the evaluation of finite element matrices. *SIAM J. Sci. Comput. 28,* 1, 224–240.

KIRBY, R. C. AND SCOTT, L. R. 2007. Geometric optimization of the evaluation of finite element matrices. *SIAM J. Sci. Comput. 29,* 2, 827–841.

SHERWIN, S. J. AND KARNIADAKIS, G. E. 2005. *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press.

SLAGLE, J. R. 1963. A heuristic program that solves symbolic integration problems in freshman calculus. *J. ACM 10,* 4, 507–520.

TERPSTRA, D., JAGODE, H., YOU, H., AND DONGARRA, J. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Springer Berlin Heidelberg, 157–173.

WANG, P. S. 1986. FINGER: A symbolic system for automatic generation of numerical programs in finite element analysis. *J. Symb. Comput. 2*, 305–316.

WOLF, M. M. AND HEATH, M. T. 2009. Combinatorial optimization of matrix-vector multiplication in finite element assembly. *SIAM J. Sci. Comput. 31,* 4, 2960–2980.

ØLGAARD, K. B. AND WELLS, G. N. 2010. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software 37,* 1, 8:1–8:23.