Optimality and Heuristics for Finite Element Integration

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

We tackle the problem of automatically generating optimal finite element integration routines given a high level specification of arbitrary multilinear forms. Optimality is defined in terms of floating point operations given a memory bound. We provide an approach to explore the transformation space of loops and expressions typical of integration routines and discuss the conditions for optimality. A theoretical analysis and extensive experimentation, which shows systematic performance improvements over a number of alternative code generation systems, validate the approach.

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

General Terms: Design, Performance

Additional Key Words and Phrases: Finite element integration, local assembly, compilers, performance optimization

ACM Reference Format:
Fabio Luporini, David A. Ham, and Paul H. J. Kelly, 2015. Optimality and Heuristics for Finite Element Integration. *ACM Trans. Arch. & Code Opt.* V, N, Article A (January YYYY), 21 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The need for rapid implementation of high performance, robust, and portable finite element methods has led to approaches based on automated code generation. This has been proven successful in the context of the FEniCS ([Logg et al. 2012]) and Firedrake ([Firedrake contributors 2014]) projects. In these frameworks, the weak variational form of a problem is expressed at high level by means of a domain-specific language. The mathematical specification is manipulated by a form compiler that generates a representation of assembly operators. By applying these operators to an element in the discretized domain, a local matrix and a local vector, which represent the contributions of that element to the equation approximated solution, are computed. The code for assembly operators must be carefully optimized: as the complexity of a variational form increases, in terms of number of derivatives, pre-multiplying functions, or poly-

This research is partly funded by the MAPDES project, by the Department of Computing at Imperial College London, by EPSRC through grants EP/I00677X/1, EP/I006761/1, and EP/L000407/1, by NERC grants NE/K008951/1 and NE/K006789/1, by the U.S. National Science Foundation through grants 0811457 and 0926687, by the U.S. Army through contract W911NF-10-1-000, and by a HiPEAC collaboration grant. The authors would like to thank Mr. Andrew T.T. McRae, Dr. Lawrence Mitchell, and Dr. Francis Russell for their invaluable suggestions and their contribution to the Firedrake project.
Author's addresses: Fabio Luporini & Paul H. J. Kelly, Department of Computing, Imperial College London; David A. Ham, Department of Computing and Department of Mathematics, Imperial College London;

nomial order of the chosen function spaces, the operation count increases, with the result that assembly often accounts for a significant fraction of the overall runtime.

As demonstrated by the substantial body of research on the topic, automating the generation of such high performance implementations poses several challenges. This is a result of the complexity inherent to the mathematical expressions involved in the numerical integration, which varies from problem to problem, and the particular structure of the loop nests enclosing the integrals. General-purpose compilers, such as *GNU's* and *Intel's*, fail at exploiting the structure inherent in the expressions, thus producing sub-optimal code (i.e., code which performs more floating-point operations, or "flops", than necessary; we show this is Section 6). Research compilers, for instance those based on polyhedral analysis of loop nests such as PLUTO ([Bondhugula et al. 2008]), focus on parallelization and optimization for cache locality, treating issues orthogonal to the question of minimising flops. The lack of suitable third-party tools has led to the development of a number of domain-specific code transformation (or synthesizer) systems. In Ølgaard and Wells [2010], it is shown how automated code generation can be leveraged to introduce optimizations that a user should not be expected to write "by hand". In Kirby and Logg [2006] and Russell and Kelly [2013], mathematical reformulations of finite element integration are studied with the aim of minimizing the operation count. In Luporini et al. [2015], the effects and the interplay of generalized code motion and a set of low level optimizations are analysed. It is also worth mentioning an on-going effort to produce a new form compiler, called UFLACS ([Alnæs 2015]), which adds to the already abundant set of code transformation systems for assembly operators. The performance evaluation in Section 6 includes most of these optimization systems.

However, in spite of such a considerable research effort, still there is no answer to one fundamental question: can we automatically generate an implementation of a form which is optimal in the number of flops executed? In this paper, we formulate an approach to solve this problem. Summarizing, our contributions are as follows:

— We define optimality for finite element integration as number of flops given a memory bound; we build the space of legal transformations within which the optimal is sought.
— We provide an algorithm to select optimal points in the transformation space. The algorithm uses a cost model to: 1) understand whether a transformation reduces or increases the operation count; 2) choose between different (non-composable) transformations.
— We integrate our approach with a compiler, COFFEE[1], which is in use in the Firedrake framework.
— We experimentally evaluate using a broader suite of forms, discretizations, and code generation systems than has been used in prior research. This is essential to demonstrate that our optimality model holds in practice.

In addition, in order to place COFFEE on the same level as other code generation systems from the viewpoint of low level optimization (which is essential for a fair performance comparison)

— We introduce an engine based on symbolic execution that allows skipping irrelevant floating point operations (e.g., those involving zero-valued quantities).

[1]COFFEE stands for COmpiler For Fast Expression Evaluation. The compiler is open-source and available at https://github.com/coneoproject/COFFEE

2. PRELIMINARIES

We review finite element integration using the same notation and examples adopted in Ølgaard and Wells [2010] and Russell and Kelly [2013].

We consider the weak formulation of a linear variational problem

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

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

$$Au = b \tag{2}$$

in which A and b discretize a and L respectively:

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

The matrix A and the vector b are “assembled” and subsequently used to solve the linear system through (typically) an iterative method.

We focus on the assembly phase, which is often characterized as a two-step procedure: *local* and *global* assembly. Local assembly is the subject of this article. It consists of computing the contributions of a single element in the discretized domain to the equation approximated solution. In global assembly, such local contributions are “coupled” by suitably inserting them into A and b.

We illustrate local assembly in a concrete example, the evaluation of the local element matrix for a Laplacian operator. Consider the weighted Laplace equation

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

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

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

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

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

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

2.1. Quadrature Mode

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

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

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

2.2. Tensor Contraction Mode

Starting from Equation 7, exploiting linearity, associativity and distributivity of the involved mathematical operators, we can rewrite the expression as

$$A_{ij}^K = \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\alpha_3=1}^{n} det F_K' w_{\alpha_3} \sum_{\beta=1}^{d} \frac{X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_{K_0} \phi_{\alpha_3} \frac{\partial \phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \phi_{i_2}}{\partial X_{\alpha_2}} dX. \tag{8}$$

A generalization of this transformation has been proposed in [Kirby and Logg 2007]. Because of only involving reference element terms, the integral in the equation can be pre-evaluated and stored in temporary variables. The evaluation of the local tensor can then be abstracted as

$$A_{ij}^K = \sum_{\alpha} A_{i_1 i_2 \alpha}^0 G_K^\alpha \tag{9}$$

in which the pre-evaluated "reference tensor" $A_{i_1 i_2 \alpha}$ and the cell-dependent "geometry tensor" G_K^α are exposed.

2.3. Qualitative Comparison

Depending on form and discretization, the relative performance of the two modes, in terms of the operation count, can vary quite dramatically. The presence of derivatives or coefficient functions in the input form tends to increase the size of the geometry tensor, making the traditional quadrature mode preferable for "complex" forms. On the other hand, speed-ups from adopting tensor mode can be significant in a wide class of forms in which the geometry tensor remains "sufficiently small". The discretization, particularly the relative polynomial order of trial, test, and coefficient functions, also plays a key role in the resulting operation count.

These two modes have been implemented in the FEniCS Form Compiler ([Kirby and Logg 2006]). In this compiler, a heuristic is used to choose the most suitable mode for a given form. It consists of analysing each monomial in the form, counting the number of derivatives and coefficient functions, and checking if this number is greater than a constant found empirically ([Logg et al. 2012]). We will later comment on the efficacy of this approach (Section 4. For the moment, we just recall that one of the goals of this research is to produce a system that goes beyond the dichotomy between quadrature and tensor modes. We will reason in terms of loop nests, code motion, and code pre-evaluation, searching the entire implementation space for an optimal synthesis.

3. TRANSFORMATION SPACE

In this section, we characterize optimality for finite element integration and the transformation space that we need to explore to achieve it.

3.1. Loop Nests and Optimality

In order to make the article self-contained, we start with reviewing basic compiler terminology.

Definition 1 (Perfect and imperfect loop nests)**.** *A perfect loop nest is a loop whose body either 1) comprises only a sequence of non-loop statements or 2) is itself a perfect loop nest. If this condition does not hold, a loop nest is said to be imperfect.*

Definition 2 (Independent basic block)**.** *An independent basic block is a sequence of statements such that no data dependencies exist between statements in the block.*

We focus on perfect nests whose innermost loop body is an independent basic block. A straightforward property of this class is that hoisting invariant expressions from the innermost to any of the outer loops or the preheader (i.e., the block that precedes the entry point of the nest) is always safe, as long as any dependencies on loop indices are honored. We will make use of this property. The results of this section could also be generalized to larger classes of loop nests, in which basic block independence does not hold, although this would require refinements beyond the scope of this paper.

By mapping mathematical properties to the loop nest level, we introduce the concepts of a *linear loop* and, more generally, a (perfect) multilinear loop nest.

Definition 3 (Linear loop)**.** *A loop L defining the iteration space I through the iteration variable i, or simply L_i, is linear if in its body*

(*1*) *i appears only as an array index, and*
(*2*) *whenever an array a is indexed by i ($a[i]$), all expressions in which this appears are affine in a.*

Definition 4 (Perfect multilinear loop nest)**.** *A perfect multilinear loop nest of arity n is a perfect nest composed of n loops, in which all of the expressions appearing in the body of the innermost loop are linear in each loop L_i separately.*

Since our focus is on finite element integration, for simplicity we restrict ourselves to the following, notable subclass of perfect multilinear loop nests.

Definition 5 (Outer product loop nest)**.** *An outer product loop nest is a perfect multilinear loop nest of arity $n \leq 2$ in which all of the expressions appearing in the body of the innermost loop are summations, and every loop index occurs once in every summand.*

As shown later, outer product loop nests are important because they have a structure that we can take advantage of to synthesize optimal code. We will see that they arise naturally when translating bilinear or linear forms into code.

Two other classes of loops need be characterized.

Definition 6 (Reduction loop)**.** *A loop L_i is said to be a reduction loop if in its body*

(*1*) *i appears only as an array index, and*
(*2*) *for each augmented assignment statement S (e.g., an increment), arrays indexed by i appear only on the right hand side of S.*

Definition 7 (Free order loop)**.** *A loop L_i is said to be a free order loop if its iterations can be executed in any arbitrary order; that is, there are no loop-carried dependencies across different iterations.*

We use these definitions to formalize the class of loop nests for which we seek optimality.

Definition 8 (Finite element integration loop nest). *A finite element integration loop nest is a loop nest in which we identify, in order, an imperfect free order loop, a (generally) imperfect, linear or non-linear reduction loop, and an outer product loop nest whose body is an independent basic block.*

To contextualize, consider Equation 7 and the (abstract) loop nest implementing it illustrated in Figure 1. The imperfect nest $\Lambda = [L_e, L_i, L_j, L_k]$ comprises a free order loop L_e (over the elements in the mesh), a reduction loop L_i (along which numerical integration is performed), and an outer product loop nest $[L_j, L_k]$ (over test and trial functions). The expression F implements the operator.

```
for (e = 0; e < L; e++)
  ...
  for (i = 0; i < M; i++)
  ..
    for (j = 0; j < N; j++)
      for (k = 0; k < O; k++)
        $a_{ejk}$ += $F(...)$
```

Fig. 1: The typical loop nest implementing a bilinear form.

We emphasize that our aim is a strategy to synthesize optimal loop nests of this specific form. In particular, we characterize optimality as follows.

Definition 9 (Optimality of a loop nest). *Let Λ be a generic loop nest, and let Γ be a generic transformation function $\Gamma : \Lambda \rightarrow \Lambda'$ such that Λ' is semantically equivalent to Λ (possibly, $\Lambda' = \Lambda$). We say that $\Lambda' = \Gamma(\Lambda)$ is an optimal synthesis of Λ if the number of operations (sums, products) that it performs to evaluate the result is minimal.*

Note that Definition 9 does not take into account memory requirements. If the loop nest were memory-bound – the ratio of operations to bytes transferred from memory to the CPU being too low – then speaking of optimality would clearly make no sense. Henceforth we assume to operate in a CPU-bound regime, in which arithmetic-intensive expressions need be evaluated. In the context of finite element, this is often true for more complex multilinear forms and/or higher order elements.

To synthesize optimal implementations as in Definition 9 we need:

— a characterization of the space of legal transformations impacting the operation count
— a cost model to select the optimal point in the transformation space

In the following Sections 3.2 and 3.3, we focus on the construction of the transformation space.

3.2. Sharing Elimination

We start with introducing the fundamental notion of sharing.

Definition 10 (Sharing). *A loop L_i presents sharing if it contains at least one statement in which two (sub-)expressions depending on i are symbolically identical.*

For instance, Figure 2(a) shows a trivial outer product loop nest of arity $n = 2$ with sharing along dimension j induced by b_j. The optimal synthesis for this loop nest is provided in Figure 2(b). This example illustrates how factorization, besides reducing the number of multiplications required at a given point in the loop nest, can expose code motion opportunities.

```
for (j = 0; j < O; j++)
  for (i = 0; i < N; i++)
    $a_{ji}$ += $b_j c_i$ + $b_j d_i$
```

(a) With sharing

```
for (i = 0; i < N; i++)
  $t_i$ = $c_i + d_i$
for (j = 0; j < O; j++)
  for (i = 0; i < N; i++)
    $a_{ji}$ += $b_j t_i$
```

(b) Optimal form

Fig. 2: Reducing a simple outer product loop nest to optimal form.

In this section, we study an optimal transformation to perform *sharing elimination*; that is, a transformation that minimizes the operation count of a loop nest by applying factorization, code motion and (if necessary) common sub-expression elimination. In general, transformations of this sort require solving large combinatorial problems; for instance, to evaluate the impact of all possible factorization strategies. However, we show that by restricting ourselves to the class of loops and expressions of Definition 8, the search space can drastically be pruned. In the following, we assume outer product loop nests of arity $n = 2$ (i.e., a bilinear form), although the same reasoning stands for the case $n < 2$.

From Definition 8 descends the fact that expressions can always be reduced to the form $\sum_{w=1}^{m} \alpha_{eij}^{w} \beta_{eik}^{w} \sigma_{ei}^{w}$, where m (the number of "terms") depends on the specific form and discretization, α_{eij} (β_{eik}) represents the product of the inverse Jacobian matrix for the change of coordinates with test (trial) functions, and σ_{ei} is a function of coefficients and geometry. We make the following considerations:

(1) Assume $\alpha \neq \beta$; that is, test and trial functions belong to different function spaces. By factorizing, in sequence, any $a_{ij} \in \alpha_{eij}$ and then any $b_{ik} \in \beta_{eik}$ (or vice-versa), sub-expressions independent of one of the outer product loops are exposed. This systematically exposes all code motion opportunities to minimize the operation count in the innermost loop.
(2) In the special case $\alpha = \beta$, common sub-expressions arise over different iteration spaces (ij and ik). Hoisting and capturing these sub-expressions may make the operation count of outer loops lower than in point 1), but the gain (if any) will be negligible (note: the operation count in the innermost loop does not vary).
(3) Since loop trip counts are never lower than 3 (P1 Lagrange elements on triangles), it is never the case that applying points 1) or 2) increases the operation count (it might have been the case, otherwise, in a scenario in which loops degenerate, i.e. with iteration spaces of size 1).
(4) Factorizing any of the terms in σ_{ei} (e.g., a coefficient, or a coefficient derivative along a certain spatial dimension) can have multiple implications. We distinguish two cases:
    (a) Factorization occurs in place of points 1) or 2). This either (i) inhibits (decrease) code motion by producing non-factorizable, non-hoistable sub-expressions or (ii) simply reduces to cases 1) or 2) (for example, in the case of forms with scalar coefficients).
    (b) Factorization occurs after points 1) or 2); for example, we have $a_{ij}(b_{ik}(\sigma'_{ei}+\sigma''_{ei}))$, with $\sigma_{ei} = \sigma'_{ei} + \sigma''_{ei}$. Splitting L_e-dependent from L_i-dependent sub-expressions (through suitable re-factorization) and applying code motion naturally tends to operation count minimization. If the loop trip count is extremely low, however, this strategy may be sub-optimal. For example, consider the following sub-expression, $(av_i + bw_i)(cv_i + dw_i)$. It is true that factorizing v_i and w_i would expose L_i-independent sub-expressions, but this would actually result in increased operation count if, for instance, the size of L_i were 1.

Based on these observations, in Algorithm 1 we explain the sequence of transformations for potentially optimal sharing elimination. The algorithm does not take into account the concerns raised in points 2) and 4b). To address them, the simplest solution consists of comparing the operation counts by generating the corresponding code variants. We will further comment on this in Section 4.3.

Algorithm 1 (Construction of the potential optimum)**.** Consider a finite element integration loop nest $\Lambda = [L_e, L_i, L_j, L_k]$. The algorithm starts with reducing the input expression to the normal form $\sum_{w=1}^{m} \alpha_{eij}^{w} \beta_{eik}^{w} \sigma_{ei}^{w}$.

The sub-expressions $\alpha_{eij}^{w}\beta_{eik}^{w}$ are "flattened" (by multiplication expansion) such that all L_j- and L_k-dependent terms lie on the same level of the expression tree. We factorize, in sequence, L_j- (L_k-) and L_k- (L_j-) dependent terms (it can easily be demonstrated, by contradiction, that one should start from the loop with the least number of unique terms in order to obtain the smallest operation count). Thanks to linearity, each factored product only has one symbol depending on L_j (L_k), and this symbol is now unique in the expression. The other sub-expression, independent of L_j (L_k), is, by definition, loop-invariant, and therefore hoisted such that redundant computation is avoided. Note that multilinearity ensures deterministic factorization, while perfectness ensures correctness of hoisting. Finally, we factorize L_i-dependent terms such that L_e-dependent sub-expressions are hoisted.

3.3. Pre-evaluation

Sharing elimination reduces the operation count by applying three operators: factorization, code motion and/or common sub-expression elimination. A fourth operator impacting the operation count is *reduction pre-evaluation*. In this section, we discuss its role and show that there exists a single point in the whole transformation space (i.e., a specific factorization) that makes applying this operator legal.

We start with an example. Consider again the bilinear form implementation in Figure 1. We pose the following question: are we able to identify sub-expressions within F for which the reduction imposed by L_i can be pre-evaluated, thus obtaining a decrease in operation count proportional to the size of L_i, M? The transformation we look for is exemplified in Figures 3(a) and 3(b); Figure 3(a), which represents the input, can be seen as a simple instance of the abstract loop nest in Figure 1.

```
for (e = 0; e < L; e++)
  for (i = 0; i < M; i++)
    for (k = 0; k < O; k++)
      $a_{ek}$ += $d_e b_{ik} c_i$ + $d_e b_{ik} d_i$
```

(a) With reduction

```
for (i = 0; i < M; i++)
  for (k = 0; k < O; k++)
    $t_k$ += $b_{ik}$($c_i$ + $d_i$)

for (e = 0; e < L; e++)
  for (k = 0; k < O; k++)
    $a_{ek}$ = $d_e t_k$
```

(b) Pre-evaluated reduction

Fig. 3: Exposing (through factorization) and pre-evaluating a reduction.

Pre-evaluation opportunities can be exposed through exploration of the expression tree transformation space. This would be challenging if we were to deal with arbitrary loop nests and expressions. However, we can make use of a result – the foundation of tensor contraction mode – to simplify our task. As summarized in Section 2.2, multilinear forms can be seen as sums of monomials, each monomial being an integral over the equation domain of products (of derivatives) of functions from discrete spaces; such monomials can always be reduced to a product of two tensors. This result can be turned into a transformation algorithm for loops and expressions, for which we provide a succinct description below.

Algorithm 2 (Pre-evaluation)**.** Consider a finite element integration loop nest $\Lambda = [L_e, L_i, L_j, L_k]$. We dissect `F` into distinct sub-expressions (the monomials). Each sub-expression is factorized so as to split constants from $[L_i, L_j, L_k]$-dependent terms. This transformation is feasible, as a consequence of the results in Kirby and Logg [2007]. These $[L_i, L_j, L_k]$-dependent terms are hoisted outside of Λ and stored into temporaries. As part of this process, the reduction induced by L_i is evaluated. Consequently, L_i disappears from Λ.

The pre-evaluation of a monomial introduces some critical issues:

(1) In contrast to what happens with hoisting in outer product loop nests, the temporary variable size is proportional to the number and trip counts of non-reduction loops crossed (for the bilinear form implementation in Figure 1, NO for sub-expressions depending on $[L_i, L_j, L_k]$ and LNO for those depending on $[L_e, L_i, L_j, L_k]$). This might shift the loop nest from a CPU-bound to a memory-bound regime, which might be counter-productive for actual execution time.
(2) The transformations exposing $[L_i, L_j, L_k]$-dependent terms increase, in general, the arithmetic complexity (e.g., expansion may increase the operation count). This could outweigh the gain due to pre-evaluation.
(3) The need for a strategy to coordinate sharing elimination and pre-evaluation opportunities: sharing elimination inhibits pre-evaluation, whereas pre-evaluation generally expose further sharing elimination opportunities.

We expand on point 1) in the next section. We address points 2) and 3) in Section 4.

3.4. Memory constraints

In the previous section, we provided an insight into the potentially negative effects of code motion. In this section, we expand on this matter. The following two observations, in particular, lead to the definition of *memory constraints*.

— The fact that $L \gg M, N, O$ suggests we should be cautious about hoisting mesh-dependent (i.e., L_e-dependent) expressions. Imagine Λ is enclosed in a time stepping loop. One could think of exposing (through some transformations) and hoisting any time-invariant sub-expressions to minimize redundant computation at every time step. The working set size could then increase by a factor L. The gain in number of operations executed could therefore be outweighed, from a runtime viewpoint, by a much larger memory pressure.
— For certain forms and discretizations, aggressive hoisting can make the working set exceed the size of some level of local memory (e.g. the last level of private cache on a conventional CPU, the shared memory on a GPU). For example, as explained in Section 3.3, pre-evaluating geometry-independent expressions outside of Λ requires temporary arrays of size NO for bilinear forms and of size N (or O) for linear forms. This can sometimes break such a "local memory threshold". In our experiments (Section 6.2) we will carefully study this aspect.

Based on these considerations, we establish the following memory constraints.

Constraint 1. *The size of a temporary due to code motion must not be proportional to the size of L_e.*

Constraint 2. *The total amount of memory occupied by the temporaries due to code motion must not exceed a certain threshold, T_H.*

Constraint 1 reflects the policy decision that the compiler should not silently consume memory on ... data objects. Consequently, generalized code motion as performed by sharing elimination is not allowed to hoist expressions outside of L_e.

We conclude providing a more refined definition of optimality.

Definition 11 (Optimality of a loop nest with bounded working set)**.** *Let Λ be a generic loop nest, and let Γ be a generic transformation function $\Gamma : \Lambda \to \Lambda'$ such that Λ' is semantically equivalent to Λ (possibly, $\Lambda' = \Lambda$) and Constraints 1 and 2 are satisfied. We say that $\Lambda' = \Gamma(\Lambda)$ is an optimal synthesis of Λ if the number of operations (sums, products) that it performs to evaluate the result is minimal.*

3.5. Completeness

We defined sharing elimination and pre-evaluation as high level transformations on top of basic operators such as code motion and factorization. Factorization addresses *spatial redundancy*. The presence of spatial redundancy means that some operations are needlessly executed at two points in an expression. Code motion and reduction pre-evaluation, on the other hand, target *temporal redundancy*; that is, the needless execution of the same operation with the same operands at two points in the loop nest.

Sharing elimination and reduction pre-evaluation tackle the problem of minimizing spatial and temporal redundancy in the loop nests and expressions of Definition 8. The space of all possible factorizations is explored, and for each point in this space, code motion (restricted by Constraint 1), common sub-expression elimination, and reduction pre-evaluation opportunities (restricted by Constraint 2) are analyzed.

4. OPTIMAL SYNTHESIS

In this section, we build a transformation algorithm capable of reaching the goal set by Definition 11 for the class of loop nests in Definition 8. In the perspective of integrating the transformation algorithm with a code generation system, we also discuss an approach that trades optimality for compilation time.

4.1. Transformation Algorithm

At this point, two main issues are to be addressed:

(1) *Coordinating the application of pre-evaluation and sharing elimination.* Recall from Section 3.3 that pre-evaluation, which is legal only under a specific factorization of the expression, could either increase or decrease the operation count with respect to sharing elimination.
(2) *Finding the global optimum.* Consider a form comprising two monomials $m1$ and $m2$. Assume that pre-evaluation is profitable for $m1$ but not for $m2$, and that $m1$ and $m2$ share at least one term (e.g. some basis functions). If pre-evaluation were applied to $m1$, sharing between $m1$ and $m2$ would be lost. We then need a mechanism to understand what transformation – pre-evaluation or sharing elimination – results in the highest operation count reduction when considering the whole set of monomials (i.e., the expression as a whole).

Let $\theta : M \to \mathbb{Z}$ be a cost function that, given a monomial $m \in M$, returns the gain/loss of pre-evaluation over sharing elimination. In particular, we have that $\theta(m) = \theta_{se}(m) - \theta_{pre}(m)$. θ_{se} and θ_{pre} represent the operation counts resulting from applying to sharing elimination and pre-evaluation, respectively. We return to the issue of constructing θ_{se} and θ_{pre} in Section 4.2. θ is used by the transformation algorithm, whose pseudo-code is provided in Figure 4.

The algorithm starts with splitting the monomials into two disjoint sets: the monomials that surely do not take advantage of pre-evaluation (S) and those that instead

```
// Initialization
M = {m | m is a monomial in the input expression}
S = {m | $\theta(m)$ < 0}
P = M \ S

// Find optimum such that memory constraints are honoured
B = {b | b is a bipartite graph from complete graph G = (P, E)}
for each b $\in$ B:
  for each (b$_S$, b$_P$) $\in$ b:
    if memory_required(b$_P$) $\geq T_H$:
      continue
    op_count[b] = $\theta_{se}$(S $\cup$ b$_S$) + $\theta_{pre}$(b$_P$)
(b$_S$, b$_P$) = min(op_count)

// Apply the transformations
pre_evaluate(P $\cup$ b$_P$)
eliminate_sharing(S $\cup$ b$_S$ $\cup$ P $\cup$ b$_P$)
```

Fig. 4: High level view of the transformation algorithm

```
// Pre-evaluated tables
...
for (e = 0; e < L; e++)
  // Temporaries due to sharing elimination
  // (Sharing was a by-product of pre-evaluation)
  ...
  // Loop nest for pre-evaluated monomials
  for (j = 0; j < N; j++)
    for (k = 0; k < O; k++)
      A[e,j,k] += G′(...) + G″(...) + ...

  // Loop nest for monomials for which run-time
  // integration was determined to be faster
  for (i = 0; i < M; i++)
    // Temporaries due to sharing elimination
    ...
    for (j = 0; j < N; j++)
      for (k = 0; k < O; k++)
        A[e,j,k] += H(...)
```

Fig. 5: The loop nest produced by the algorithm for an input as in Figure 1.

potentially benefit from it (P). Some subsets of monomials in P: 1) might share terms with S, in which case sharing elimination could globally outperform pre-evaluation; 2) might break Constraint 2. The algorithm then finds the optimal bipartition $b = (b_S, b_p)$ of P such that b_P represents the subset of monomials for which pre-evaluation is globally optimal and Constraint 2 is honoured. Since the number of monomials in a form tends to be very small (a few units in most complex forms), evaluating all possible bipartitions is not a significant issue as long as the cost of calculating θ_{se} and θ_{pre} is negligible. This aspect is elaborated in Section 4.2. In addition, observe that because of reuse of basis functions, pre-evaluation may result in identical tables, which will be mapped to the same temporary. Therefore, sharing elimination is transparently applied to all monomials (i.e., to all sub-expressions, even those produced by pre-evaluation). The output of the transformation algorithm is as in Figure 5, assuming the usual input of Figure 1.

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < O; k++)
      A[j,k] += B[i,j]*B[i,k]*(f[0]*B[i,0]+f[1]*B[i,1]+f[2]*B[i,2])
```

Fig. 6: Simplified loop nest for a pre-multiplied mass matrix.

In the following section, we tie up the remaining loose end, the construction of the cost function θ.

4.2. The cost function θ

We recall that $\theta(m) = \theta_{se}(m) - \theta_{pre}(m)$, with θ_{se} and θ_{pre} representing the operation counts after applying sharing elimination and with pre-evaluation. Since θ is expected to be used by a compiler, requirements are simplicity and velocity. In the following, we explain how to derive these two values.

The most trivial way of evaluating θ_{se} and θ_{pre} is to apply the transformations and count the number of operations. If, on one hand, this is definitely possible for θ_{se} (performing Algorithm 1 tends to have negligible cost), the code generation time would become unacceptable if we were to apply pre-evaluation (Algorithm 2, symbolic execution of L_i) to each possible bipartition. We then need an analytic way of determining θ_{pre}.

The first step consists of estimating the *increase factor*, ι. This number captures the increase in arithmetic complexity due to the transformations enabling pre-evaluation. To contextualize, consider the example in Figure 6. One can think of this as the (simplified) loop nest originating from the integration of a pre-multiplied mass matrix. The sub-expression `f[0]*B[i,0]+f[1]*B[i,1]+f[2]*B[i,2]` represents the field f over (tabulated) basis functions (array B). In order to apply pre-evaluation, the expression needs be transformed to separate f from all L_i-dependent quantities. By product expansion, we can observe an increase in the number of $[L_j, L_k]$-dependent terms of a factor 3 (the number of basis functions for f). ι captures this increase in non-hoistable operations.

With a single coefficient, as we just saw, ι directly descends from the cost of product expansion. In general, however, predicting ι is less straightforward. For example, consider the case in which a monomial has multiple coefficients over the same function space. Product expansion would lead to identical sub-expressions, and ι must reflect this. To evaluate ι, we then calculate the k-combinations with repetitions of n elements, with k being the number of coefficient-dependent terms appearing in a product (in the example, there is only f, so $k = 1$) and n the cardinality of the set of symbols involved in the coefficient expansion (in the example, `B[i,0]`, `B[i,1]`, and `B[i,2]`, so $n = 3$).

If $\iota \geq I$ we can immediately say that pre-evaluation will not be profitable. This is a necessary condition telling us that if we add to the innermost loop more operations than we actually save from eliminating L_i, then for sure $\theta_{se} < \theta_{pre}$. This observation can speed up the analysis cost.

If, on the other hand, $\iota < I$, a further step is necessary to estimate θ_{pre}. In particular, we need to calculate the number of terms ρ such that $\theta_{pre} = \rho \cdot \iota$. Consider again Figure 6. In the case of the mass matrix, the body of L_k is simply characterized by the dot product of test and trial functions, `B[j]*B[k]`, so trivially $\rho = 1$. In general, ρ varies with the discretization and the differential operators used in the form. For example, in the case of the bilinear form originating from a standard bi-dimensional Poisson equation, the reader could verify that after a suitable factorization we would have $\rho = 3$. There are several ways of determining ρ. The fastest would be to extract it from high-level analysis of form and discretization; for convenience, in our implementation we have algorithms that, based on analysis of the expression tree, project the output of monomial expansion and factorization, which in turn gives us ρ.

4.3. Trading optimality for analysis cost

— SPEED UP THETA: We can easily predict θ_{se} thanks to our key property, linearity. This was explained in Proposition **??**. A simple expression tree traversal suffices to obtain the cost of a sharing-free outer product loop nest, namely Λ_{ops}^{ns}. Assuming I to be the size of L_i, we have that $\theta_{se} = \Lambda_{ops}^{ns} \cdot I$.
— SPEED UP ALGO: most of the time, only a single bipartition...
— SPEED UP SHARING EL: stick to opt

5. CODE GENERATION

The transformation algorithm described in Section 4 has been implemented in COFFEE, the optimizer of finite element integration routines used in Firedrake. In this section, we describe several aspects related to code generation.

5.1. Automation through the COFFEE Language

COFFEE implements all optimizations by composing "building-block" transformations, which we refer to as "rewrite operators". This has several advantages. Firstly, extendibility: novel transformations – for instance, sum-factorization in spectral methods – could be expressed using the existing operators, or with small effort building on what is already available. Secondly, generality: any sort of codes characterized by (linear, but also non-linear) loop nests with sharing can be optimized (obviously, without guarantees on optimality). In other words, COFFEE only exploits domain properties, and it is not specifically tied to finite element in any way. Thirdly, robustness: the same operators are exploited, and therefore stressed, by different optimization pipelines.

The rewrite operators, which implementation is based on abstract syntax tree manipulation, compose the COFFEE language. A non-exhaustive list of such operators includes expansion, factorization, re-association, generalized code motion. Sharing elimination and pre-evaluation are implemented by applying (composing) in specific sequences these operators.

5.2. Independence from Form Compilers

One of the key pillars of COFFEE is independence from form compilers. Being capable of handling generic abstract syntax trees, COFFEE can be leveraged by any suitably adapted form compiler. For example, in Firedrake a modified version of the FEniCS Form Compiler producing abstract syntax trees instead of strings is used. COFFEE itself provides an interface for building such abstract syntax trees. In particular, COFFEE aims to decouple the form manipulation (i.e., the "math level") from code optimization (flops executed, low level optimization such as vectorization). Another viewpoint is: developers of form compilers should not worry about expression optimization in any way, since this is handled at a lower level.

To achieve this goal, COFFEE needs a strategy to minimize the execution of potentially "useless" flops due to simplistic implementations of non scalar-valued function spaces. For example, the most trivial quadrature mode in the FEniCS Form Compiler implements vector-valued function spaces by populating tabulated basis functions with blocks of zero-valued columns. For the local element matrix/vector evaluation, these blocks are iterated over in a single, rectangular iteration space. This preserves the semantics of the computation while keeping the implementation complexity extremely low, although it also introduces unnecessary operations (e.g., zero-valued summands due to multiplications by zero). COFFEE achieves full independence from form compilers (and so relives the implementation burden) by providing a transformation to avoid the execution of such useless flops. A challenge for this transformation is restructuring the iteration spaces while preserving the effectiveness of low level optimizations, especially (auto-)vectorization.

Consider a set of tabulated basis functions with quadrature points along rows and functions along columns. For example, `A[i,j]` provides the value of the `j-th` basis function at quadrature point `i`. In Ølgaard and Wells [2010], a technique to avoid iteration over zero-valued columns based on the use of indirection arrays (e.g. `A[B[i]]`, in which `A` is a tabulated basis function and `B` a map from loop iterations to non-zero columns in A) was proposed. Our approach aims to avoid such indirection arrays in the generated code. This is because we want to avoid non-contiguous memory loads and stores, which can nullify the benefits of vectorization.

The idea is that if the dimension along which vectorization is performed (typically the innermost) has a contiguous slice of zeros, but that slice is smaller than the vector length, then we do nothing (i.e., the loop nest is not transformed). Otherwise, we restructure the iteration space. This has several non-trivial implications. The most notable one is memory offsetting (e.g., `A[i+m,j+n]`), which dramatically enters in conflict with padding and data alignment. We use heuristics to retain the positive effects of both and to ensure correctness. Details are, however, beyond the scope of this paper.

The implementation is based on symbolic execution: the loop nests are traversed and for each statement encountered the location of zeros in each of the involved symbols is tracked. Arithmetic operators have a different impact on tracking. For example, multiplication requires computing the set intersection of the zero-valued slices (for each loop dimension), whereas addition requires computing the set union.

6. PERFORMANCE EVALUATION

6.1. Experimental Setup

Experiments were run on a single core of an Intel I7-2600 (Sandy Bridge) CPU, running at 3.4GHz, 32KB L1 cache (private), 256KB L2 cache (private) and 8MB L3 cache (shared). The Intel Turbo Boost and Intel Speed Step technologies were disabled. The Intel `icc 15.2` compiler was used. The compilation flags used were `-O3, -xHost, -ip`.

We analyze the runtime performance of four real-world bilinear forms of increasing complexity, which comprise the differential operators that are most common in finite element methods. In particular, we study the mass matrix ("`Mass`") and the bilinear forms arising in a Helmholtz equation ("`Helmholtz`"), in an elastic model ("`Elasticity`"), and in a hyperelastic model ("`Hyperelasticity`"). The complete specification of these forms is made publicly available[2].

We evaluate the speed-ups achieved by a wide variety of transformation systems over the "original" code produced by the FEniCS Form Compiler (i.e., no optimizations applied). We analyze the following transformation systems

— FEniCS Form Compiler: optimized quadrature mode (work presented in Ølgaard and Wells [2010]). Referred to as `quad`
— FEniCS Form Compiler: tensor mode (work presented in Kirby and Logg [2006]). Referred to as `tens`
— FEniCS Form Compiler: automatic mode (choice between `tens` and `quad` driven by heuristic, detailed in Logg et al. [2012] and summarized in Section 2.3). Referred to as `auto`
— UFLACS: a novel back-end for the FEniCS Form Compiler (whose primary goals are improved code generation time and runtime). Referred to as `ufls`

[2]https://github.com/firedrakeproject/firedrake-bench/blob/experiments/forms/firedrake_forms.py

— COFFEE: generalized loop-invariant code motion (work presented in Luporini et al. [2015]). Referred to as `cf01`
— COFFEE: optimal loop nest synthesis plus symbolic execution for zero-elimination (work of this article). Referred to as `cf02`

The values that we report are the average of three runs with "warm cache" (no code generation time, no compilation time). They include the cost of local assembly as well as the cost of matrix insertion. However, the unstructured mesh used for the simulations (details below) was chosen small enough to fit the L3 cache of the CPU so as to minimize the "noise" due to operations outside of the element matrix evaluation.

For a fair comparison, small patches (publicly available) were written to run *all* simulations through Firedrake. This means the costs of matrix insertion and mesh iteration are identical in all variants. Our patches make UFLACS and the FEniCS Form Compiler's optimization systems generate code suitable for Firedrake, which employs a data storage layout different than that of FEniCS (e.g., array of pointers instead of pointer to pointers).

In Section 3.4, we discussed the importance of memory constraints. We then define T_H as the maximum amount of space that temporaries due to code motion can take. We set $T_H = L2_{size}$, that is, the size of the processor L2 cache (the last level of private cache). We recall that exceeding this threshold prevents the application of pre-evaluation. In our experiments, this happened in some circumstances. In such cases, experiments were repeated with $T_H = L3_{size}$ to verify the hypotheses made in Section 3.4. We later elaborate on this.

Following the methodology adopted in Ølgaard and Wells [2010], we vary the following parameters:

— the polynomial order of test, trial, and coefficient (or "pre-multiplying") functions, $q \in \{1, 2, 3, 4\}$
— the number of coefficient functions $nf \in \{0, 1, 2, 3\}$

While constants of our study are

— the space of test, trial, and coefficient functions: Lagrange
— the mesh: tetrahedral with a total of 4374 elements
— exact numerical quadrature (we employ the same scheme used in Ølgaard and Wells [2010], based on the Gauss-Legendre-Jacobi rule)

6.2. Performance Results

We report the results of our experiments in Figures 7, 8, 9, and 10 as three-dimensional plots. The axes represent q, nf, and code transformation system. We show one subplot for each problem instance $\langle form, nf, q\rangle$, with the code transformation system varying within each subplot. The best variant for each problem instance is given by the tallest bar, which indicates the maximum speed-up over non-transformed code. We note that if a bar or a subplot are missing, then the form compiler failed at generating code because of either exceeding the system memory limit or unable to handle the form.

The rest of the section is structured as follows: we provide insights about the main message of the experimentation; we comment on the impact of autovectorization; we explain in detail, individually for each form, the performance results obtained.

High level view. The main observation is that our transformation strategy does not always guarantee minimum execution time. In particular, 5% of the test cases (3 out of 56, without counting marginal differences) show that `cf02` was not optimal in terms of runtime. The most significant of such test cases is the elastic model with $[q = 4, nf = 0]$. There are two reasons for this. Firstly, low level optimization can have a significant im-

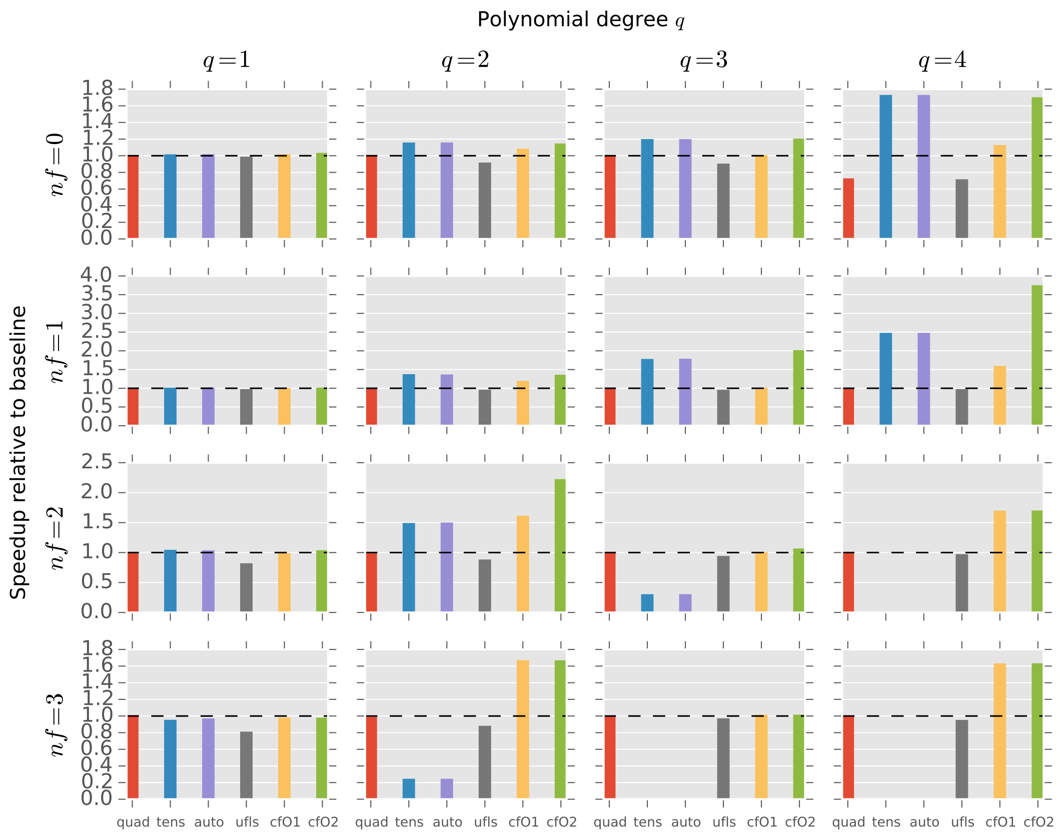

Fig. 7: Performance evaluation for the *mass* matrix. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

pact on actual performance. For example, the aggressive loop unrolling in `tens` eliminates operations on zeros and reduces the working set size by not storing entire temporaries; on the other hand, preserving the loop structure can maximize the chances of autovectorization. Secondly, memory constraints are critical, particularly the transformation strategy adpopted when exceeding T_H. We will later thoroughly elaborate on all these aspects.

Autovectorization. The discretizations employed result in inner loops and basis function tables of size multiple of the machine vector length. This, combined with the chosen compilation flags, promotes autovectorization in the majority of code variants. An exception is quad due to the presence of indirection arrays in the generated code. In `tens`, loop nests are fully unrolled, so the standard loop vectorization is not feasible; manual inspection of the compiled code suggests, however, that block vectorization ([Larsen and Amarasinghe 2000]) is often triggered. In `ufls`, `cfO1`, and `cfO2` the iteration spaces have similar structure (there are a few exceptions in `cfO2` due to zero-elimination), with loop vectorization being regularly applied, as far as we could evince from compiler reports and manual inspection of assembly code.

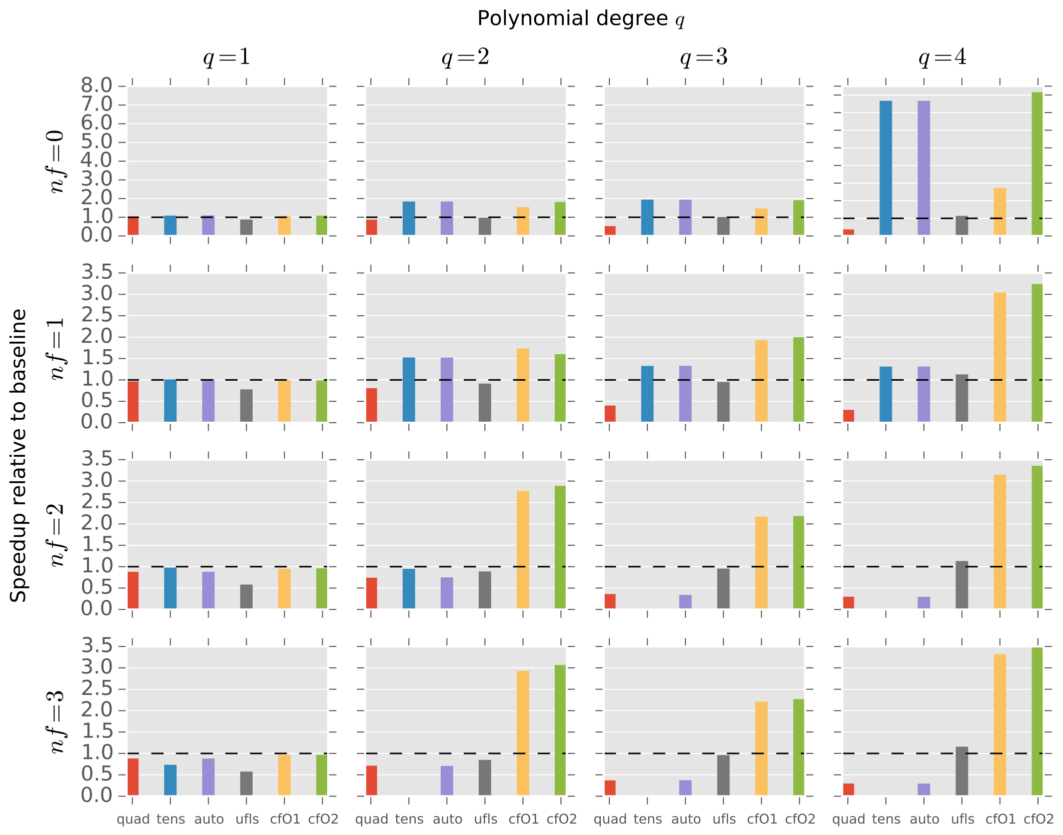

Fig. 8: Performance evaluation for the bilinear form of a *Helmholtz* equation. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

Mass. We start with the simplest of the bilinear forms investigated, the mass matrix. Results are in Figure 7. We first notice that the lack of improvements when $q = 1$ is due to the fact that matrix insertion outweighs local assembly. As $q \geq 2$, `cfO2` generally shows the highest speed-ups. It is worth noting how `auto` does not always select the fastest implementation: `auto` always opts for `tens`, while as $nf \geq 2$ `quad` would tend to be preferable. On the other hand, `cfO2` always makes the optimal decision about whether applying pre-evaluation or not.

Helmholtz. As happened with the mass matrix problem, when $q = 1$ matrix insertion still hides the cost of local assembly. For $q \geq 2$, the general trend is that `cfO2` outperforms the competitors. In particular, with

— $nf = 0$, the adoption of pre-evaluation by `cfO2` results in increasingly notable speed-ups over `cfO1`, as q increases; `tens` is comparable to `cfO2`, with `auto` making the right choice.

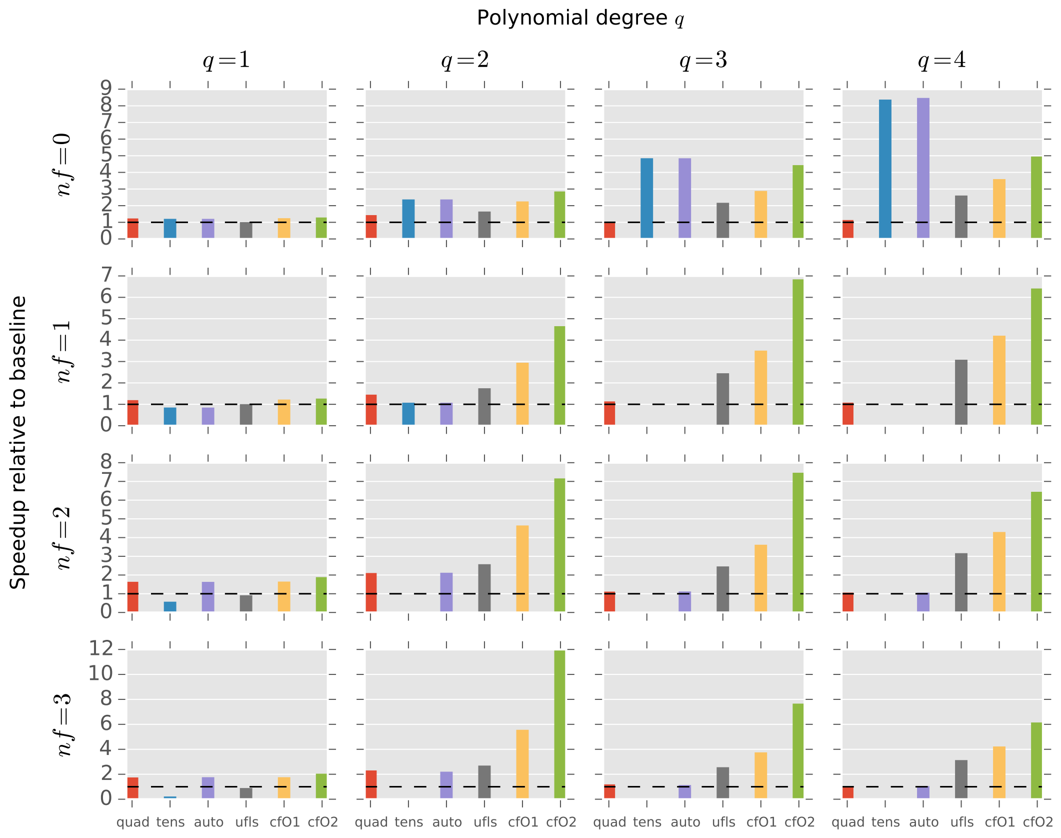

Fig. 9: Performance evaluation for the bilinear form arising in an *elastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

— $nf = 1$, auto picks tens; the choice is however sub-optimal when $q = 3$ and $q = 4$. This can indirectly be inferred from the large gap between cf01/cf02 and tens/auto: cf02 applies sharing elimination, but it avoids pre-evaluation.
— $nf = 2$ and $nf = 3$, auto reverts to quad, which would theoretically be the right choice (the flop count is much lower than in tens or what would be produced by pre-evaluation); however, the generated code suffers from the presence of indirection arrays, which break autovectorization and "traditional" code motion.

The sporadic slow-downs or only marginal improvements exhibited by ufls are imputable to the presence of sharing.

An interesting experiment we performed was relaxing the memory threshold by setting it to $T_H = L3_{size}$. We found that this makes cf02 generally slower as $nf \geq 2$, with a maximum slow-down of 2.16× with $\langle nf = 2, q = 2\rangle$. The effects of not having a sensible threshold could even be worse in parallel runs, since the L3 cache is shared by the cores.

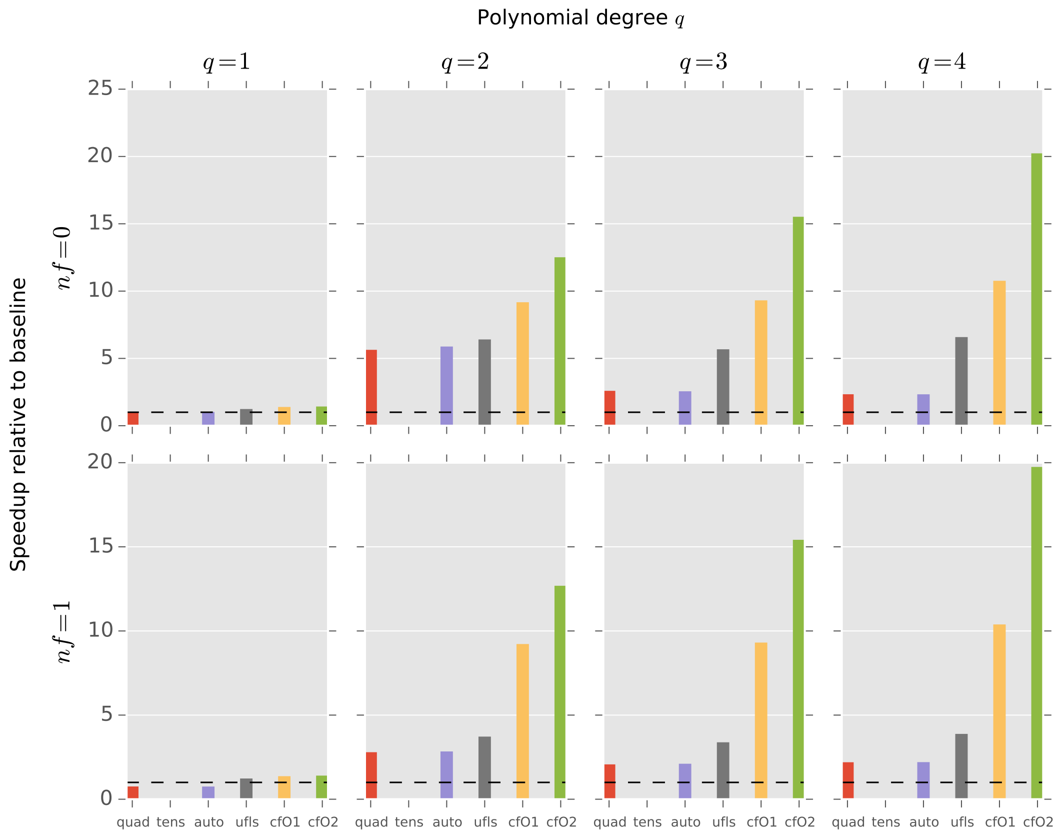

Fig. 10: Performance evaluation for the bilinear form arising in a *hyperelastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

Elasticity. The results for the elastic model are displayed in Figure 9. The main observation is that cfO2 never triggers pre-evaluation, although in some occasions it should. To clarify this, consider the test case $\langle nf = 0, q = 2\rangle$, in which tens/auto show a considerable speed-up over cfO2. cfO2 finds pre-evaluation profitable – that is, actually capable of reducing the operation count – although it does not apply it because otherwise T_H would be exceeded. However, running the same experiments with $T_H = L3_{size}$ resulted in a dramatic improvement, even higher than that of tens. Our explanation is that despite exceeding T_H by roughly 40%, the save in operation count is so large ($5\times$ in this problem) that pre-evaluation would anyway be the optimal choice. This suggests that our model could be refined to handle the cases in which there is a significant gap between potential cache misses and save in flops.

We also note that:

— the differences between cfO2 and cfO1 are due to systematic sharing elimination and the use of symbolic execution to avoid iteration over the zero-valued regions in the basis function tables

— when $nf = 1$, `auto` prefers `tens` to `quad`, which leads to sub-optimal operation counts and execution times
— `ufls` generally shows better runtime behaviour than `quad` and `tens`. This is due to multiple facts, including avoidance of indirection arrays, preservation of loop structure, a more effective code motion.

Hyperelasticity. In the experiments on the hyperelastic model, shown in Figure 10, `cf02` exhibits the largest gains out of all problem instances considered in this paper. This is a positive aspect: it means that our transformation algorithm scales with form complexity. The fact that all code transformation systems (apart from `tens`) show quite significant speed-ups suggests several points. Firstly, the baseline is highly inefficient: with forms as complex as in this hyperelastic model, a trivial translation of integration routines into code should always be avoided since even one of the best general-purpose compilers available (Intel's on an Intel platform at maximum optimization level) is not capable of exploiting the structure inherent in the mathematical expressions generated. Secondly, the code motion strategy really makes a considerable impact. The sharing elimination performed by `cf02` in each level of the loop nest ensures a critical reduction in operation count, which results is better execution times. In particular, at higher order, the main difference between `ufls` and `cf02` is due to the application of this transformation to the outer product loop nest. Clearly, the operation count increases with q, and so do the speed-ups.

7. CONCLUSIONS

With this research we have set the foundation of optimal finite element integration. We have developed theory and implemented an automated system capable of applying it. The automated system, COFFEE, is integrated in Firedrake, a real-world framework for writing finite element methods. We believe the results are extremely positive. An open problem is understanding how to optimally handle non-linear loop nests. A second open problem is extending our methodology to classes of loops arising in spectral methods; here, the interaction with low level optimization will probably become stronger due to the typically larger working sets deriving from the use of high order function spaces. Lastly, we recall our work is publicly available and is already in use in the latest version of the Firedrake framework.

REFERENCES

Martin Sandve Alnæs. 2015. UFLACS - UFL Analyser and Compiler System. https://bitbucket.org/fenics-project/uflacs. (2015).

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

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

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

Robert C. Kirby and Anders Logg. 2007. Efficient Compilation of a Class of Variational Forms. *ACM Trans. Math. Softw.* 33, 3, Article 17 (Aug. 2007). DOI:http://dx.doi.org/10.1145/1268769.1268771

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

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

Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. 2015. Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly. *ACM Trans. Archit. Code Optim.* 11, 4, Article 57 (Jan. 2015), 25 pages. DOI:http://dx.doi.org/10.1145/2687415

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

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