A

An algorithm for the optimization of finite element integration loops

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

We present an algorithm for the optimization of a class of finite element integration loop nests. This algorithm, which exploits fundamental mathematical properties of finite element operators, is proven to achieve a locally optimal operation count. In specified circumstances the optimum achieved is global. Extensive numerical experiments demonstrate significant performance improvements over the state of the art in finite element code generation in almost all cases. This validates the effectiveness of the algorithm presented here, and illustrates its limitations.

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, 2016. An algorithm for the optimization of finite element integration loops. *ACM Trans. Math. Softw.* V, N, Article A (January YYYY), 26 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 [Rathgeber et al. 2015] projects. In these frameworks, the weak variational form of a problem is expressed in a high level mathematical syntax by means of the domain-specific language UFL [Alnæs et al. 2014]. This mathematical specification is used by a domain-specific compiler, known as a form compiler, to generate low-level C or C++ code for the integration over a single element of the computational mesh of the variational problem's left and right hand side operators. 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 polynomial order of the

This work was supported by by the Department of Computing at Imperial College London, the Engineering and Physical Sciences Research Council [grant number EP/L000407/1], and the Natural Environment Research Council [grant numbers NE/K008951/1 and NE/K006789/1], and by a HiPEAC collaboration grant. The authors would like to thank Dr. 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 Mathematics, Imperial College London;

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

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 in 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 those by *GNU* and *Intel*, fail to exploit 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 in Section 7). 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. Ølgaard and Wells [2010] show how automated code generation can be leveraged to introduce optimizations that a user should not be expected to write "by hand". Kirby and Logg [2006] and Russell and Kelly [2013] employ mathematical reformulations of finite element integration 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 two new new form compilers, UFLACS [Alnæs 2016] and TSFC [Homolya and Mitchell 2016], which particularly target the compilation time challenges of the more complex variational forms. The performance evaluation in Section 7 includes most of these systems.

However, in spite of such a considerable research effort, there is still 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 that solves this problem for a particular class of forms and provides very good approximations in all other cases. In particular, we will define "local optimality", which relates operation count with inner loops. In summary, our contributions are as follows:

— We formalize the class of finite element integration loop nests and we build the space of legal transformations impacting their operation count.
— We provide an algorithm to select points in the transformation space. The algorithm uses a cost model to: (i) understand whether a transformation reduces or increases the operation count; (ii) choose between different (non-composable) transformations.
— We demonstrate that our approach systematically leads to a local optimum. We also explain under what conditions of the input problem global optimality is achieved.
— 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):

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

— We introduce a transformation based on symbolic execution that allows irrelevant floating point operations to be skipped (for example those involving zero-valued quantities).

After reviewing basic concepts in finite element integration, in Section 3.1 we introduce a set of definitions mapping mathematical properties to the level of loop nests. This step is an essential precursor to the definition of the two algorithms – sharing elimination (Section 3.2) and pre-evaluation (Section 3.3) – through which we construct the space of legal transformations. The main transformation algorithm in Section 4 delivers the local optimality claim by using a cost model to coordinate the application of sharing elimination and pre-evaluation. We elaborate on the correctness of the methodology in Section 5. The numerical experiments are showed in Section 7. We conclude discussing the limitations of the algorithms presented and future work.

2. PRELIMINARIES

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

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 suitable 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's approximated solution. During global assembly, these 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 Poisson 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 basis functions with support on

the element K (i.e. those which do not vanish on this element), 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. Monomials

It has been shown (for example in Kirby and Logg [2007]) that local element tensors can be expressed as a sum of integrals over K, each integral being the product of derivatives of functions from sets of discrete spaces and, possibly, functions of some spatially varying coefficients. An integral of this form is called *monomial*.

2.2. 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 to real coordinates $x = F_K(X)$ is necessary any time a new element is evaluated. The basis functions $\{\phi_i^K\}$ are then replaced with local basis functions $\{\Phi_i\}$ such that $\Phi_i(X) = \phi_i^k(F_K(X)) = \phi_i^k(x)$. The numerical integration of (6) over an element K can then be expressed as follows:

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

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

2.3. Tensor contraction mode

By exploiting the linearity, associativity and distributivity of the relevant mathematical operators, we can rewrite (7) as

$$A_{ij}^K = \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\alpha_3=1}^{n} \left(\det(F_K') w_{\alpha_3} \sum_{\beta=1}^{d} \frac{X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \left(\sum_{q=1}^{N} \Phi_{\alpha_3} \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} W^q \right) \right). \tag{8}$$

A generalization of this transformation was introduced in [Kirby and Logg 2007]. Since it only involves reference element terms, the quadrature sum can be pre-evaluated and reused for each element. The evaluation of the local tensor can then be abstracted as

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

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

2.4. 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 increases the rank of the geometry tensor, making the traditional quadrature mode preferable for sufficiently 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 polynomial order of trial, test, and coefficient functions, also plays a key role in the resulting operation count.

These two modes are 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 return to the efficacy of this approach in section 7. One of the objectives of this paper 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 global and local optimality for finite element integration as well as the space of legal transformations that needs be explored to achieve them. The method by which exploration is performed is discussed in Section 4.

3.1. Loop nests, expressions 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 $a[i]$.*

Definition 4 (Multilinear loop nest)**.** *A 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 affine in each loop L_i separately.*

We will show that multilinear loop nests, which arise naturally when translating bilinear or linear forms into code, are important because they have a structure that we can take advantage of to reach a local optimum.

We define two other classes of loops.

```
for (e = 0; e < E; e++)
  ...
  for (i = 0; i < I; i++)
    ...
    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        a_ejk += sum_{w=1}^{m} α^w_eij β^w_eik σ^w_ei
```

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

Definition 5 (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 6 (Order-free loop)**.** *A loop L_i is said to be an order-free loop if its iterations can be executed in any arbitrary order.*

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 an order-free loop L_e (over elements in the mesh), a reduction loop L_i (performing numerical integration), and a multilinear loop nest $[L_j, L_k]$ (over test and trial functions). In the body of L_k, one or more statements evaluate the local tensor for the element e. Expressions (the right hand side of a statement) result from the translation of a form in high level matrix notation into code. In particular, m is the number of monomials (a form is a sum of monomials), α_{eij} (β_{eik}) represents the product of a coefficient function (e.g., the inverse Jacobian matrix for the change of coordinates) with test or trial functions, and σ_{ei} is a function of coefficients and geometry. We do not pose any restrictions on function spaces (e.g., scalar- or vector-valued), coefficient expressions (linear or non-linear), differential and vector operators, so σ_{ei} can be arbitrarily complex. We say that such an expression is in *normal form*, because the algebraic structure of a variational form is intact: products have not yet been expanded, distinct monomials can still be identified, and so on. This brings us to formalize the class of loop nests that we aim to optimize.

Definition 7 (Finite element integration loop nest)**.** *A finite element integration loop nest is a loop nest in which the following appear, in order: an imperfect order-free loop, an imperfect (perfect only in some special cases), linear or non-linear reduction loop, and a multilinear loop nest whose body is an independent basic block in which expressions are in normal form.*

We then characterize optimality for a finite element integration loop nest as follows.

Definition 8 (Optimality of a loop nest)**.** *Let Λ be a generic loop nest, and let Γ be a transformation function $\Gamma : \Lambda \to \Lambda'$ 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 total number of operations (additions, products) performed to evaluate the result is minimal.*

The concept of local optimality, which relies on the particular class of *flop-decreasing* transformations, is also introduced.

Definition 9 (Flop-decreasing transformation)**.** *A transformation which reduces the operation count is called flop-decreasing.*

Definition 10 (Local optimality of a loop nest)**.** *Given Λ, Λ' and Γ as in Definition 8, we say that $\Lambda' = \Gamma(\Lambda)$ is a locally optimal synthesis of Λ if:*

— *the number of operations (additions, products) in the innermost loops performed to evaluate the result is minimal, and*
— Γ *is expressed as composition of flop-decreasing transformations.*

The restriction to flop-decreasing transformations aims to exclude those apparent optimizations that, to achieve flop-optimal innermost loops, would rearrange the computation at the level of the outer loops causing, in fact, a global increase in operation count.

We also observe that Definitions 8 and 10 do not take into account memory requirements. If the execution of loop nest were memory-bound – the ratio of operations to bytes transferred from memory to the CPU being too low – then optimizing the number of flops would be fruitless. Henceforth we assume we operate in a CPU-bound regime, evaluating arithmetic-intensive expressions. In the context of finite elements, this is often true for more complex multilinear forms and/or higher order elements.

Achieving optimality in polynomial time is not generally feasible, since the σ_{ei} sub-expressions can be arbitrarily unstructured. However, multilinearity results in a certain degree of regularity in α_{eij} and β_{eik}. In the following sections, we will elaborate on these observations and formulate an approach that achieves: (i) at least a local optimum in all cases; (ii) global optimality whenever the monomials are "sufficiently structured". To this purpose, we will construct:

— the space of legal transformations impacting the operation count (Sections 3.2 – 3.4)
— an algorithm to select points in the transformation space (Section 4)

3.2. Sharing elimination

We start with introducing the fundamental notion of sharing.

Definition 11 (Sharing)**.** *A statement within a loop nest Λ presents sharing if at least one of the following conditions hold:*

Spatial sharing. There are at least two symbolically identical sub-expressions
Temporal sharing. There is at least one non-trivial sub-expression (e.g., an addition or a product) that is redundantly executed because it is independent of $\{L_{i_1}, L_{i_1}, ...L_{i_n}\} \subset \Lambda$.

To illustrate the definition, we show in Figure 2 how sharing evolves as factorization and code motion are applied to a trivial multilinear loop nest. In the original loop nest (Figure 2(a)), spatial sharing is induced by the symbol b_j. Factorization eliminates spatial sharing and creates temporal sharing (Figure 2(b)). Finally, generalized code motion [Luporini et al. 2015], which hoists sub-expressions that are redundantly executed by at least one loop in the nest[2], leads to optimality (Figure 2(c)).

```
for (j = 0; j < J; j++)
  for (i = 0; i < I; i++)
    a_ji += b_j c_i + b_j d_i
```

(a) With spatial sharing

```
for (j = 0; j < J; j++)
  for (i = 0; i < I; i++)
    a_ji += b_j(c_i + d_i)
```

(b) With temporal sharing

```
for (i = 0; i < I; i++)
  t_i = c_i + d_i
for (j = 0; j < J; j++)
  for (i = 0; i < I; i++)
    a_ji += b_j t_i
```

(c) Optimal form

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

[2]Traditional loop-invariant code motion, which is commonly applied by general-purpose compilers, only checks invariance with respect to the innermost loop.

In this section, we study *sharing elimination*, a transformation that aims to reduce the operation count by removing sharing through the application of expansion, factorization, and generalized code motion. If the objective were reaching optimality and the expressions lacked structure, a transformation of this sort would require solving a large combinatorial problem – for instance to evaluate the impact of all possible factorizations. Our sharing elimination strategy, instead, exploits the structure inherent in finite element integration expressions to guarantee, after coordination with other transformations (an aspect which we discuss in the following sections), local optimality. Global optimality is achieved if stronger preconditions hold. Setting local optimality, rather than optimality, as primary goal is essential to produce simple and computationally efficient algorithms – two necessary conditions for integration with a compiler.

3.2.1. Identification and exploitation of structure. Finite element expressions can be seen as composition of operations between tensors. Often, the optimal implementation strategy for these operations is to be determined out of two alternatives. For instance, consider $J^{-T}\nabla v \cdot J^{-T}\nabla v$, with J^{-T} being the transposed inverse Jacobian matrix for the change of (two-dimensional) coordinates, and v a generic two-dimensional vector. The tensor operation will reduce to the scalar expression $(av_i^0 + bv_i^1)(av_i^0 + bv_i^1) + ...$, in which v_i^0 and v_i^1 represent components of v that depend on L_i. To minimize the operation count for expressions of this kind, we have two options:

Strategy 1. *Eliminating temporal sharing through generalized code motion.*

Strategy 2. *Eliminating spatial sharing first – through product expansion and factorization – and temporal sharing afterwards, again through generalized code motion.*

In the current example, we observe that, depending on the size of L_i, applying Strategy 2 could reduce the operation count since the expression would be recast as $v_i^0 v_i^0 aa + v_i^0 v_i^1 (ab + ab) + v_i^1 v_i^1 cc + ...$ and some hoistable sub-expressions would be exposed. On the other hand, Strategy 1 would have no effect as v only depends on a single loop, L_i. In general, the choice between the two strategies depends on multiple factors: the loop sizes, the increase in operation count due to expansion (in Strategy 2), and the gain due to code motion. A second application of Strategy 2 was provided in Figure 2. These examples motivate the introduction of a particular class of expressions, for which the two strategies assume notable importance.

Definition 12 (Structured expression)**.** *We say that an expression is "structured along a loop nest Λ" if and only if, for every symbol s_Λ depending on at least one loop in Λ, the spatial sharing of s_Λ may be eliminated by factorizing all occurrences of s_Λ in the expression.*

Proposition 1. *An expression along a multilinear loop nest is structured.*

Proof. This follows directly from Definition 3 and Definition 4, which essentially restrict the number of occurrences of a symbol s_Λ in a summand to at most 1. □

If Λ were an arbitrary loop nest, a given symbol s_Λ could appear everywhere (e.g., n times in a summand and m times in another summand with $n \neq m$, as argument of a higher level function, in the denominator of a division), thus posing the challenge of finding the factorization that maximizes temporal sharing. If Λ is instead a finite element integration loop nest, thanks to Proposition 1 the space of flop-decreasing transformations is constructed by "composition" of Strategy 1 and Strategy 2, as illustrated in Algorithm 1.

Finally, we observe that the σ_{ei} sub-expressions can sometimes be considered "weakly structured". This happens when a relaxed version of Definition 12 applies,

in which the factorization of s_Λ only "minimizes" (rather than "eliminates") spatial sharing (for instance, in the complex hyperelastic model analyzed in Section 7). Weak structure will be exploited by Algorithm 1 in the attempt to achieve optimality.

3.2.2. Algorithm. Algorithm 1 describes sharing elimination assuming as input a tree representation of the loop nest. It makes use of the following notation and terminology:

— *multilinear operand*: any α_{eij} or β_{eik} in the input expression.
— *multilinear symbol*: a symbol appearing within a multilinear operand depending on L_j or L_k (e.g., test functions, first order derivatives of test functions, etc.).

Examples will be provided in Section 3.2.3.

Algorithm 1 (Sharing elimination)**.** The input of the algorithm is a tree representation a finite element integration loop nest.

(1) Perform a depth-first visit of the loop tree to collect and partition multilinear operands into disjoint sets, $\mathbb{P} = \{P_1, ..., P_p\}$. $\mathbb{P}$ is such that all multilinear operands in each $P \in \mathbb{P}$ share the same set of multilinear symbols S_P, whereas there is no sharing across different partitions. For all multilinear operands in $P \in \mathbb{P}$ such that $|P| \leq |S_P|$, apply Strategy 1.
Note: as a consequence of Proposition 1, $|P|$ and $|S_P|$ represent the number of products in the innermost loop induced by P if Strategy 1 or Strategy 2 were applied
(2) For each sub-expression expr depending on exactly one linear loop, collect the multilinear symbols and the temporaries produced at step (1). Partition them into disjoint sets, $\mathbb{T} = \{T_1, ..., T_t\}$, such that T_i includes all instances of a given symbol in expr. Apply Strategy 2 factorizing the symbols in each T_i, provided that this leads to a reduction in operation count; otherwise, apply Strategy 1
Note: the last check ensures the flop-decreasing nature of the transformation. In the cases in which expansion outweighs code motion, Strategy 1 is preferred.
Note: the expansion cost is a function of the products wrapping a symbol (how many of them and their arity), so it can be determined through tree visits.
(3) Build the *sharing graph* $G = (S, E)$. Each $s \in S$ represents a multilinear symbol or a temporary produced by the previous steps. An edge (s_i, s_j) indicates that a product $s_i s_j$ would appear if the sub-expressions including s_i and s_j were expanded.
Note: the following steps will only impact bilinear forms, since otherwise $E = \emptyset$.
(4) Partition S into disjoint sets, $\mathbb{S} = \{S_1, ..., S_n\}$, such that S_i includes all instances of a given symbol s in the expression. Transform G by merging $\{s_1, ..., s_m\} \subset S_i$ into a unique vertex s (taking the union of the edges), provided that factorizing $[s_1, ..., s_m]$ would not cause an increase in operation count.
(5) Map G to an Integer Linear Programming (ILP) model for determining how to optimally apply Strategy 2. The solution is the set of symbols that will be factorized by Strategy 2. Let $|S| = n$; the ILP model then is as follows:

$$x_i\text{: a vertex in } S \text{ (1 if a symbol should be factorized, 0 otherwise)}$$
$$y_{ij}\text{: an edge in } E \text{ (1 if } s_i \text{ is factorized in the product } s_i s_j \text{ , 0 otherwise)}$$
$$n_i\text{: the number of edges incident to } x_i$$

$$\min \sum_{i=1}^{n} x_i,\ s.t. \qquad \sum_{j|(i,j)\in E} y_{ij} \leq n_i x_i,\ i = 1, ..., n$$
$$y_{ij} + y_{ji} = 1,\ (i,j) \in E$$

(6) Perform a depth-first visit of the loop tree and, for each yet unhandled or hoisted expression, apply the most profitable between Strategy 1 and Strategy 2.
Note: this pass speculatively assumes that expressions are (weakly) structured along the reduction loop. If the assumption does not hold, the operation count will generally be sub-optimal because only a subset of factorizations and code motion opportunities may eventually be considered.

Although the primary goal of Algorithm 1 is operation count minimization within the multilinear loop nest, the enforcement of flop-decreasing transformations (steps (2) and (4)) and the re-scheduling of sub-expressions within outer loops (last step) also attempt to optimize the loop nest globally. We will further elaborate this aspect in Section 5.

3.2.3. Examples. Consider again Figure 2(a). We have $\mathbb{P} = \{P_0, P_1, P_2\}$, with $P_0 = \{b_j\}$, $P_1 = \{c_i\}$, and $P_2 = \{d_i\}$. For all P_i, we have $|P_i| = 1 = |S_{P_i}|$, although applying Strategy 1 in step (1) has no effect. The sharing graph is $G = (\{b_j, c_i, d_i\}, \{(b_j, c_i), (b_j, d_i)\}$, and $T = \{c_i, d_i\}$. The ILP formulation leads to the code in Figure 2(c).

In Figure 3, Algorithm 1 is executed in a very simple realistic scenario, which originates from the bilinear form of a Poisson equation in two dimensions. We observe that $\mathbb{P} = \{P_0, P_1\}$, with $P_0 = \{(z_0 a_{ik} + z_2 b_{ik}), (z_1 a_{ik} + z_3 b_{ik})\}$ and $P_1 = \{(z_0 a_{ij} + z_2 b_{ij}), (z_1 a_{ij} + z_3 b_{ij})\}$. In addition, $|P_i| = |S_{P_i}| = 2$, so Strategy 1 is applied to both partitions (step (1)). We then have (step (3)) $G = (\{t_0, t_1, t_2, t_3\}, \{(t_0, t_2), (t_1, t_3)\})$. Since there are no more factorization opportunities, the ILP formulation becomes irrelevant.

```
for (e = 0; e < E; e++)
  z0 = ...
  z1 = ...
  ...
  for (i = 0; i < I; i++)
    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        a_ejk += (((z0*a_ik + z2*b_ik)*
                   (z0*c_ij + z2*d_ij))+
                  ((z1*a_ik + z3*b_ik)*
                   (z1*c_ij + z3*d_ij)))*
                 W_i * det
```

(a) Normal form

```
for (e = 0; e < E; e++)
  ...
  for (i = 0; i < I; i++)
    for (k = 0; k < K; k++)
      t0_k = (z0*a_ik + z2*b_ik)
      t1_k = (z1*a_ik + z3*b_ik)
    for (j = 0; j < J; j++)
      t2_j = (z0*c_ij + z2*d_ij) * W_i * det
      t3_j = (z1*c_ij + z3*d_ij) * W_i * det
    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        a_ejk += t0_k * t2_j + t1_k * t3_j
```

(b) After sharing elimination

Fig. 3: Applying sharing elimination to the bilinear form arising from a Poisson equation in 2D. The operation counts are $E(f(z_0, z_1, ...) + IJK \cdot 18)$ (left) and $E(f(z_0, z_1, ...) + I(J \cdot 6 + K \cdot 9 + JK \cdot 4)$ (right), with $f(z_0, z_1, ...)$ representing the operation count for evaluating $z_0, z_1, ...$, and common sub-expressions being counted once. The synthesis in Figure 3(b) is globally optimal apart from the pathological case $I, J, K = 1$.

For reasons of space, further examples, including the hyperelastic model evaluated in Section 7 and other non-trivial ILP instances, are made available online[3].

[3]Sharing elimination examples: https://gist.github.com/FabioLuporini/14e79457d6b15823c1cd

3.3. Pre-evaluation of reductions

Sharing elimination uses three operators: expansion, factorization, and code motion. In this section, we discuss the role and legality of a fourth operator: reduction pre-evaluation. We will see that what makes this operator special is the fact that there exists a single point in the transformation space of a monomial (i.e., a specific factorization of test, trial, and coefficient functions) ensuring its correctness.

We start with an example. Consider again the loop nest and the expression in Figure 1. We pose the following question: are we able to identify sub-expressions for which the reduction induced by L_i can be pre-evaluated, thus obtaining a decrease in operation count proportional to the size of L_i, I? The transformation we look for is exemplified in Figure 4 with a simple loop nest. The reader may verify that a similar transformation is applicable to the example in Figure 3(a).

```
for (e = 0; e < E; e++)
  for (i = 0; i < I; i++)
    for (k = 0; k < K; k++)
      a_ek += d_e b_ik c_i + d_e b_ik d_i
```

(a) With reduction

```
for (i = 0; i < I; i++)
  for (k = 0; k < K; k++)
    t_k += b_ik (c_i + d_i)

for (e = 0; e < E; e++)
  for (k = 0; k < K; k++)
    a_ek = d_e t_k
```

(b) After pre-evaluation

Fig. 4: Exposing (through factorization) and pre-evaluating a reduction.

Pre-evaluation can be seen as the generalization of tensor contraction (Section 2.3) to a wider class of sub-expressions. We know that 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. A monomial can always be reduced to the product between a "reference" and a "geometry" tensor. In our model, a reference tensor is simply represented by one or more sub-expressions independent of L_e, exposed after particular transformations of the expression tree. This leads to the following algorithm.

Algorithm 2 (Pre-evaluation)**.** Consider a finite element integration loop nest $\Lambda = [L_e, L_i, L_j, L_k]$. We dissect the normal form input expression into distinct sub-expressions, each of them representing a monomial. Each sub-expression is then factorized so as to split constants from $[L_i, L_j, L_k]$-dependent terms. This transformation is feasible[4], 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 computed by means of symbolic execution. Finally, L_i is removed from Λ.

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

(1) Depending on the complexity of a monomial, a certain number, t, of temporary variables is required if pre-evaluation is performed. Such temporary variables are actually n-dimensional arrays of size S, with n and S being, respectively, the arity and

[4]For reasons of space, we omit the detailed sequence of steps (e.g., expansion, factorization), which is however available at https://github.com/coneoproject/COFFEE/blob/master/coffee/optimizer.py in Luporini et al. [2016].

the extent (iteration space size) of the multilinear loop nest (e.g., $n = 2$ and $S = JK$ in the case of bilinear forms). For certain values of $<t, n, S>$, pre-evaluation may dramatically increase the working set, which may be counter-productive for actual execution time.
(2) The transformations exposing $[L_i, L_j, L_k]$-dependent terms increase the arithmetic complexity of the expression (e.g., expansion tends to increase the operation count). This could outweigh the gain due to pre-evaluation.
(3) A strategy for coordinating sharing elimination and pre-evaluation is needed. We observe that sharing elimination inhibits pre-evaluation, whereas pre-evaluation could expose further sharing elimination opportunities.

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

3.4. Memory constraints

We have just observed that the code motion induced by monomial pre-evaluation may dramatically increase the working set size. Even more aggressive code motion strategies are theoretically conceivable. Imagine Λ is enclosed in a time stepping loop. One could think of exposing (through some transformations) and hoisting time-invariant sub-expressions for minimizing redundant computation at each time step. The working set size would then increase by a factor E, and since $E \gg I, J, K$, the gain in operation count would probably be outweighed, from a runtime viewpoint, by a much larger memory pressure.

Since, for certain forms and discretizations, hoisting may cause the working set to 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), we introduce 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 is a policy decision that the compiler should not silently consume memory on global data objects. It has the effect of shrinking the transformation space. Constraint 2 has both theoretical and practical implications, which will be carefully analyzed in the next sections.

4. SELECTION AND COMPOSITION OF TRANSFORMATIONS

In this section, we build a transformation algorithm that, given a memory bound, systematically reaches a local optimum for finite element integration loop nests.

4.1. Transformation algorithm

We address the two following issues:

(1) *Coordination of pre-evaluation and sharing elimination.* Recall from Section 3.3 that pre-evaluation could either increase or decrease the operation count in comparison with that achieved by sharing elimination.
(2) *Optimizing over composite operations.* Consider a form comprising two monomials m_1 and m_2. Assume that pre-evaluation is profitable for m_1 but not for m_2, and that m_1 and m_2 share at least one term (for example some basis functions). If pre-evaluation were applied to m_1, sharing between m_1 and m_2 would be lost. We then need a mechanism to understand which 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 achieved by pre-evaluation over sharing elimination. In particular, we define $\theta(m) = \theta^{pre}(m) - \theta^{se}(m)$, where θ^{se} and θ^{pre} represent the operation counts resulting from applying sharing elimination and pre-evaluation, respectively. Thus pre-evaluation is profitable for m if and only if $\theta(m) < 0$. We return to the issue of deriving θ^{se} and θ^{pre} in Section 4.2. Having defined θ, we can now describe the transformation algorithm (Algorithm 3).

Algorithm 3 (Transformation algorithm)**.** The algorithm has three main phases: initialization (step 1); determination of the monomials preserving the memory constraints that should be pre-evaluated (steps 2-4); application of pre-evaluation and sharing elimination (step 5).

(1) Perform a depth-first visit of the expression tree and determine the set of monomials M. Let S be the subset of monomials m such that $\theta(m) > 0$. The set of monomials that will *potentially* be pre-evaluated is $P = M \setminus S$.
Note: there are two fundamental reasons for not pre-evaluating $m_1 \in P$ straight away: 1) the potential presence of spatial sharing between m_1 and $m_2 \in S$, which impacts the search for the global optimum; 2) the risk of breaking Constraint 2.
(2) Build the set B of all possible bipartitions of P. Let D be the dictionary that will store the operation counts of different alternatives.
(3) Discard $b = (b_S, b_P) \in B$ if the memory required after applying pre-evaluation to the monomials in b_P exceeds T_H (see Constraint 2); otherwise, add $D[b] = \theta^{se}(S \cup b_S) + \theta^{pre}(b_P)$.
Note: $\mathbb{B}$ is in practice very small, since even complex forms usually have only a few monomials. This pass can then be accomplished rapidly as long as the cost of calculating θ^{se} and θ^{pre} is negligible. We elaborate on this aspect in Section 4.2.
(4) Take $\arg\min_b D[b]$.
(5) Apply pre-evaluation to all monomials in b_P. Apply sharing elimination to all resulting expressions.
Note: because of the reuse of basis functions, pre-evaluation may produce some identical tables, which will be mapped to the same temporary variable. Sharing elimination is therefore transparently applied to all expressions, including those resulting from pre-evaluation.

The output of the transformation algorithm is provided in Figure 5, assuming as input the loop nest in Figure 1.

4.2. The cost function θ

We tie up the remaining loose end: the construction of 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 pre-evaluation. Since θ is deployed in a working compiler, simplicity and efficiency are essential characteristics. In the following, we explain how to derive these two values.

The most trivial way of evaluating θ^{se} and θ^{pre} would consist of applying the actual transformations and simply count the number of operations. This would be tolerable for θ^{se}, as Algorithm 1 tends to have negligible cost. However, the overhead would be unacceptable if we applied pre-evaluation – in particular, symbolic execution – to all bi-

```
// Pre-evaluated tables
...
for (e = 0; e < E; e++)
  // Temporaries due to sharing elimination
  // (Sharing was a by-product of pre-evaluation)
  ...
  // Loop nest for pre-evaluated monomials
  for (j = 0; j < J; j++)
    for (k = 0; k < K; k++)
      a_ejk += F'(...) + F''(...) + ...

  // Loop nest for monomials for which run-time
  // integration was determined to be faster
  for (i = 0; i < I; i++)
    // Temporaries due to sharing elimination
    ...
    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        a_ejk += H(...)
```

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

partitions analyzed by Algorithm 3. We therefore seek 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 exposing pre-evaluation opportunities. For context, consider the example in Figure 6. One can think of this as the (simplified) loop nest originating from the integration of the action of a mass matrix. The sub-expression f_0*B_{i0}+f_1*B_{i1}+f_2*B_{i2} represents the coefficient f over (tabulated) basis functions (array B). In order to apply pre-evaluation, the expression needs be transformed to separate f from all $[L_i, L_j, L_k]$-dependent quantities (see Algorithm 2). By product expansion, we observe an increase in the number of $[L_j, L_k]$-dependent terms of a factor $\iota = 3$.

```
for (i = 0; i < I; i++)
  for (j = 0; j < J; j++)
    for (k = 0; k < K; k++)
      a_jk += b_ij*b_ik*(f_0*B_i0+f_1*B_i1+f_2*B_i2)
```

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

In general, however, determining ι is not so straightforward since redundant tabulations may result from common sub-expressions. Consider the previous example. One may add one coefficient in the same function space as f, repeat the expansion, and observe that multiple sub-expressions (e.g., $b_{10}*b_{01}*...$ and $b_{01}*b_{10}*...$) will reduce to identical tables. To evaluate ι, we then use combinatorics. We calculate the k-combinations with repetitions of n elements, where: (i) k is the number of (derivatives of) coefficients appearing in a product; (ii) n is the number of unique basis functions involved in the expansion. In the original example, we had $n = 3$ (for b_{i0}, b_{i1}, and b_{i2}) and $k = 1$, which confirms $\iota = 3$. In the modified example, there are two coefficients, so $k = 2$, which means $\iota = 6$.

If $\iota \geq I$ (the extent of the reduction loop), we already know that pre-evaluation will not be profitable. Intuitively, this means that we are introducing more operations than we are saving from pre-evaluating L_i. If $\iota < I$, we still need to find the number of terms ρ such that $\theta^{pre} = \rho \cdot \iota$. The mass matrix monomial in Figure 6 is characterized by the

dot product of test and trial functions, so trivially $\rho = 1$. In the example in Figure 3, instead, we have $\rho = 3$ after a suitable factorization of basis functions. In general, therefore, ρ depends on both form and discretization. To determine this parameter, we look at the re-factorized expression (as established by Algorithm 2), and simply count the terms amenable to pre-evaluation.

5. FORMALIZATION

We demonstrate that the orchestration of sharing elimination and pre-evaluation performed by the transformation algorithm guarantees local optimality (Definition 10). The proof re-uses concepts and explanations provided throughout the paper, as well as the terminology introduced in Section 3.2.2.

Proposition 2. *Consider a multilinear form comprising a set of monomials M, and let Λ be the corresponding finite element integration loop nest. Let Γ be the transformation algorithm. Let X be the set of monomials that, according to Γ, need to be pre-evaluated, and let $Y = M \setminus X$. Assume that the pre-evaluation of different monomials does not result in identical tables. Then, $\Lambda' = \Gamma(\Lambda)$ is a local optimum in the sense of Definition 10 and satisfies Constraint 2.*

Proof. We first observe that the cost function θ predicts the *exact* gain/loss in monomial pre-evaluation, so X and Y can actually be constructed.

Let c_Λ denote the operation count for Λ and let $\Lambda_I \subset \Lambda$ be the subset of innermost loops (all L_k loops in Figure 5). We need to show that there is no other synthesis Λ''_I satisfying Constraint 2 such that $c_{\Lambda''_I} < c_{\Lambda'_I}$. This holds if and only if

(1) *The coordination of pre-evaluation with sharing elimination is optimal*. This boils down to prove that
   (a) *pre-evaluating any $m \in Y$ would result in $c_{\Lambda''_I} > c_{\Lambda'_I}$*
   (b) *not pre-evaluating any $m \in X$ would result in $c_{\Lambda''_I} > c_{\Lambda'_I}$*
(2) *Sharing elimination leads to a (at least) local optimum.*

We discuss these points separately

(1) (a) Let T_m represent the set of tables resulting from applying pre-evaluation to a monomial m. Consider two monomials $m_1, m_2 \in Y$ and the respective sets of pre-evaluated tables, T_{m_1} and T_{m_2}. If $T_{m_1} \cap T_{m_2} \neq \emptyset$, at least one table is assignable to the same temporary. Γ, therefore, may not be optimal, since θ only distinguishes monomials in "isolation". We neglect this scenario (see assumptions) because of its purely pathological nature and its – with high probability – negligible impact on the operation count.
   (b) Let $m_1 \in X$ and $m_2 \in Y$ be two monomials sharing some generic multilinear symbols. If m_1 were carelessly pre-evaluated, there may be a potential gain in sharing elimination that is lost, potentially leading to a non-optimum. This situation is prevented by construction, because Γ exhaustively searches all possible bipartitions on order to determine an optimum which satisfies Constraint 2[5]. Recall that since the number of monomials is in practice very small, this pass can rapidly be accomplished.
(2) Consider Algorithm 1. Proposition 1 ensures that there are only two ways of scheduling the multilinear operands in $P \in \mathbb{P}$: through generalized code motion (Strategy 1) or factorization of multilinear symbols (via Strategy 2). If applied, these

[5]Note that the problem can be seen as an instance of the well-known Knapsack problem

two strategies would lead, respectively, to performing $|P|$ and $|S_P|$ multiplications at every loop iteration. Since Strategy 1 is applied if and only if $|P| < |S_P|$ and does not change the structure of the expression (it requires neither expansion nor factorization), step (1) cannot prune the optimum from the search space.
After structuring the sharing graph G in such a way that only flop-decreasing transformations are possible, the ILP model is instantiated. At this point, proving optimality reduces to establishing the correctness of the model, which is relatively straightforward because of its simplicity. The model aims to minimize the operation count by selecting the most promising factorizations. The second set of constraints is to select all edges (i.e., all multiplications), exactly once. The first set of inequalities allows multiplications to be scheduled: once a vertex s is selected (i.e., once a symbol is decided to be factorized), all multiplications involving s can be grouped.

□

Throughout the paper we have reiterated the claim that Algorithm 3 achieves a globally optimal flop count if stronger preconditions on the input variational form are satisfied. We state here these preconditions, in increasing order of complexity.

(1) There is a single monomial and only one coefficient (e.g., the coordinates field). This is by far the simplest scenario, which requires no particular transformation at the level of the outer loops, so optimality naturally follows.
(2) There is a single monomial, but multiple coefficients are present. Optimality is achieved if and only if all sub-expressions depending on coefficients are structured (see Section 3.2.1). This avoids ambiguity in factorization, which in turn guarantees that the output of step (7) in Algorithm 1 is optimal.
(3) There are multiple monomials, but either at most one coefficient (e.g., the coordinates field) or multiple coefficients not inducing sharing across different monomials are present. This reduces, respectively, to cases (1) and (2) above.
(4) There are multiple monomials, and coefficients are shared across monomials. Optimality is reached if and only if the coefficient-dependent sub-expressions produced by Algorithm 1 – that is, the by-product of factorizing test/trial functions from distinct monomials – preserve structure.

6. CODE GENERATION

Sharing elimination and pre-evaluation, as well as the transformation algorithm, have been implemented in COFFEE, the compiler for finite element integration routines adopted in Firedrake. In this section, we briefly discuss the aspects of the compiler that are relevant for this article.

6.1. Expressing transformations through the COFFEE language

COFFEE implements sharing elimination and pre-evaluation by composing building block transformation operators, which we refer to as *rewrite operators*. This has several advantages. The first is extensibility. New transformations, such as sum factorization in spectral methods, could be expressed by composing the existing operators, or with small effort building on what is already available. Second, generality: COFFEE can be seen as a lightweight, low level computer algebra system, not necessarily tied to finite element integration. Third, robustness: the same operators are exploited, and therefore tested, by different optimization pipelines. The rewrite operators, whose (Python) implementation is based on manipulation of abstract syntax trees (ASTs), comprise the COFFEE language. A non-exhaustive list of such operators includes expansion, factorization, re-association, generalized code motion.

6.2. Independence from form compilers

COFFEE aims to be independent of the high level form compiler. It provides an interface to build generic ASTs and only expects expressions to be in normal form (or sufficiently close to it). For example, Firedrake has transitioned from a version of the FEniCS Form Compiler [Kirby and Logg 2006] modified to produce ASTs rather than strings, to a newly written compiler[6], while continuing to emply COFFEE. Thus, COFFEE decouples the mathematical manipulation of a form from code optimization; or, in other words, relieves form compiler developers of the task of fine scale loop optimization of generated code.

6.3. Handling block-sparse tables

For several reasons, basis function tables may be block-sparse (e.g., containing zero-valued columns). For example, the FEniCS Form Compiler implements vector-valued functions by adding blocks of zero-valued columns to the corresponding tabulations; this extremely simplifies code generation (particularly, the construction of loop nests), but also affects the performance of the generated code due to the execution of "useless" flops (e.g., operations like `a + 0`). 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. This technique, however, produces non-contiguous memory loads and stores, which nullify the potential benefits of vectorization. COFFEE, instead, handles block-sparse basis function tables by restructuring loops in such a manner that low level optimization (especially vectorization) is only marginally affected. This is based on symbolic execution of the code, which enables a series of checks on array indices and loop bounds which determine the zero-valued blocks which can be skipped without affecting data alignment.

7. PERFORMANCE EVALUATION

7.1. 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`. The compilation flag `xHost` tells the Intel compiler to generate efficient code for the underlying platform.

The Zenodo system was used to archive all packages used to perform the experiments: Firedrake [Mitchell et al. 2016], PETSc [Smith et al. 2016], petsc4py [Firedrake 2016], FIAT [Rognes et al. 2016], UFL [Alnæs et al. 2016], FFC [Logg et al. 2016], PyOP2 [Rathgeber et al. 2016b] and COFFEE [Luporini et al. 2016]. The experiments can be reproduced using a publicly available benchmark suite [Rathgeber et al. 2016a].

We analyze the execution time 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[7].

[6] TSFC, the two-stage form compiler https://github.com/firedrakeproject/tsfc
[7] https://github.com/firedrakeproject/firedrake-bench/blob/experiments/forms/firedrake_forms.py

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:

quad. Optimized quadrature mode. Work presented in Ølgaard and Wells [2010], implemented in in the FEniCS Form Compiler.
tens. Tensor contraction mode. Work presented in Kirby and Logg [2006], implemented in the FEniCS Form Compiler.
auto. Automatic choice between `tens` and `quad` driven by heuristic (detailed in Logg et al. [2012] and summarized in Section 2.4). Implemented in the FEniCS Form Compiler.
ufls. UFLACS, a novel back-end for the FEniCS Form Compiler whose primary goals are improved code generation and execution times.
cfO1. Generalized loop-invariant code motion. Work presented in Luporini et al. [2015], implemented in COFFEE.
cfO2. Optimal loop nest synthesis with handling of block-sparse tables. Work presented in this article, implemented in COFFEE.

The values that we report are the average of three runs with "warm cache"; that is, with all kernels retrieved directly from the Firedrake's cache, so code generation and compilation times are not counted. The timing includes however the cost of both local assembly and matrix insertion, with the latter minimized through the choice of a mesh (details below) small enough to fit the L3 cache of the CPU.

For a fair comparison, small patches were written to the make `quad`, `tens`, and `ufls` compatible with Firedrake. By executing all simulations in Firedrake, we guarantee that both matrix insertion and mesh iteration have a fixed cost, independent of the transformation system employed. The patches adjust the data storage layout to what Firedrake expects (e.g., by generating an array of pointers instead of a pointer to pointers, by replacing flattened arrays with bi-dimensional ones).

For Constraint 2, discussed in Section 3.4, we set $T_H = \text{size(L2)}$; that is, the size of the processor L2 cache (the last level of private cache). When the threshold had an impact on the transformation process, the experiments were repeated with $T_H = \text{size(L3)}$. The results are documented later, individually for each problem.

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

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

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)
— double-precision arithmetic

7.2. 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 $\langle \text{form}, \text{nf}, q \rangle$, 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

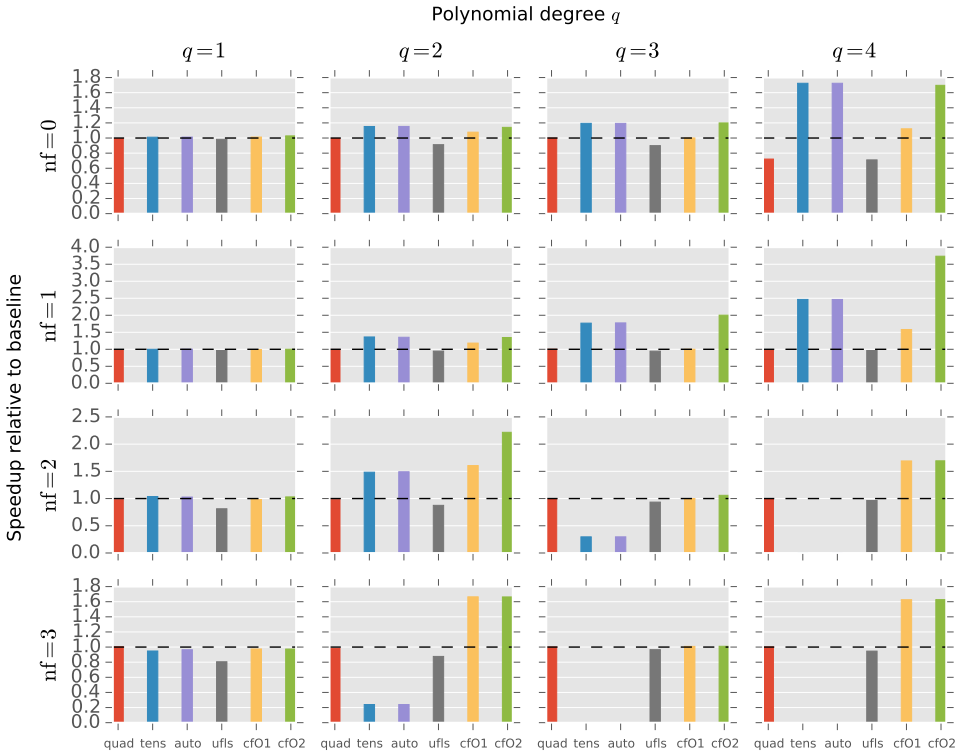

Fig. 7: Performance evaluation for the *mass* matrix. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

a bar or a subplot are missing, then the form compiler failed to generate code because it either exceeded the system memory limit or was otherwise unable to handle the form.

The rest of the section is organized as follows: we first provide insights into the general outcome of the experimentation; we then comment on the impact of a fundamental low-level optimization, namely autovectorization; finally, we motivate, for each form, the performance results obtained and the relationship with the operation count.

High level view. Our transformation strategy does not always guarantee minimum execution time. In particular, about 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 $[q = 4,\ \mathrm{nf} = 0]$. There are two reasons for this. First, low level optimization can have a significant impact on the 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. Second, the transformation strategy adopted when T_H is exceeded plays a key role, as we will later elaborate.

Autovectorization. We chose the mesh dimension and the function spaces such that the inner loop sizes would always be a multiple of the machine vector length. This

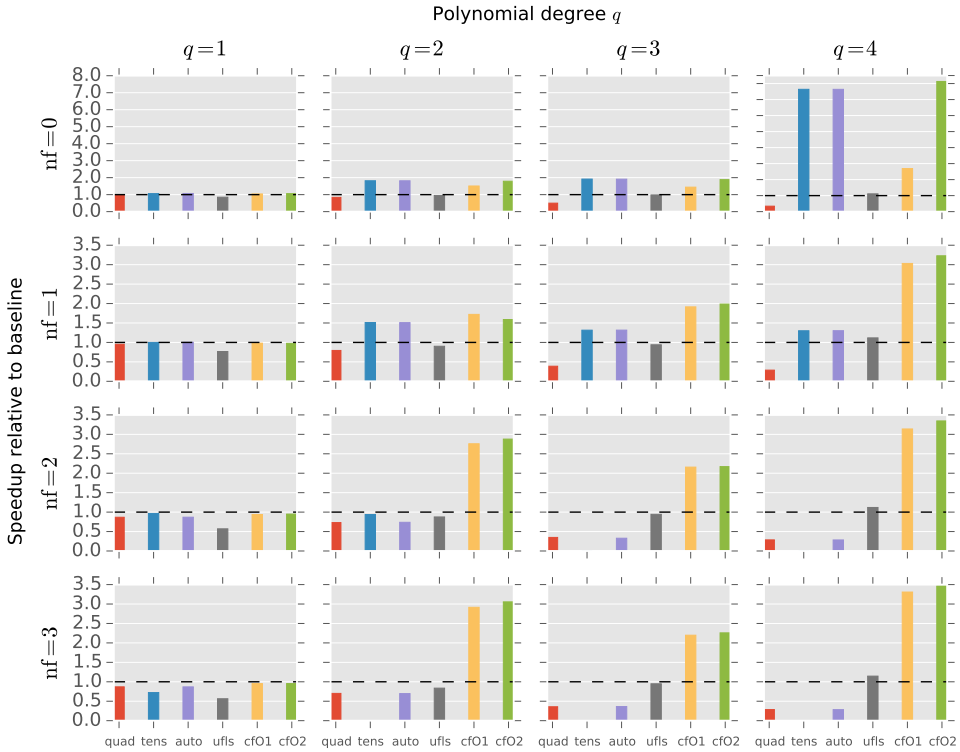

Fig. 8: Performance evaluation for the bilinear form of a *Helmholtz* equation. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

ensured autovectorization in the majority of code variants[8]. The biggest 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; the compiler reports suggest, however, that block vectorization [Larsen and Amarasinghe 2000] is often triggered. In `ufls`, `cf01`, and `cf02` the iteration spaces have identical structure, with loop vectorization being regularly applied.

Mass matrix. We start with the simplest of the bilinear forms investigated, the mass matrix. Results are in Figure 7.

We first notice the lack of improvements when $q = 1$. The operation count of `cf02` and `tensor` is roughly $9\times$ lower than that of the alternatives (tens instead of hundreads operations), but nontheless no gain in execution time is obtained. This is due to the fact that local assembly is memory-bound. For instance, in the simplest case $nf = 0$, in addition to the determinant of the Jacobian inverse, `tens` only requries three unique multiplications to evaluate the local element matrix (26 operations in total), whereas almost a hundred memory accesses are performed. In particular, we have: 12 loads for the vertex coordinates; for each vertex, 1 load to indirectly access its data; for each of the 16 local elements, 4 loads (optimistically) and 1 store to update an entry in the

[8]We verified the vectorization of inner loops by looking at both compiler reports and assembly code.

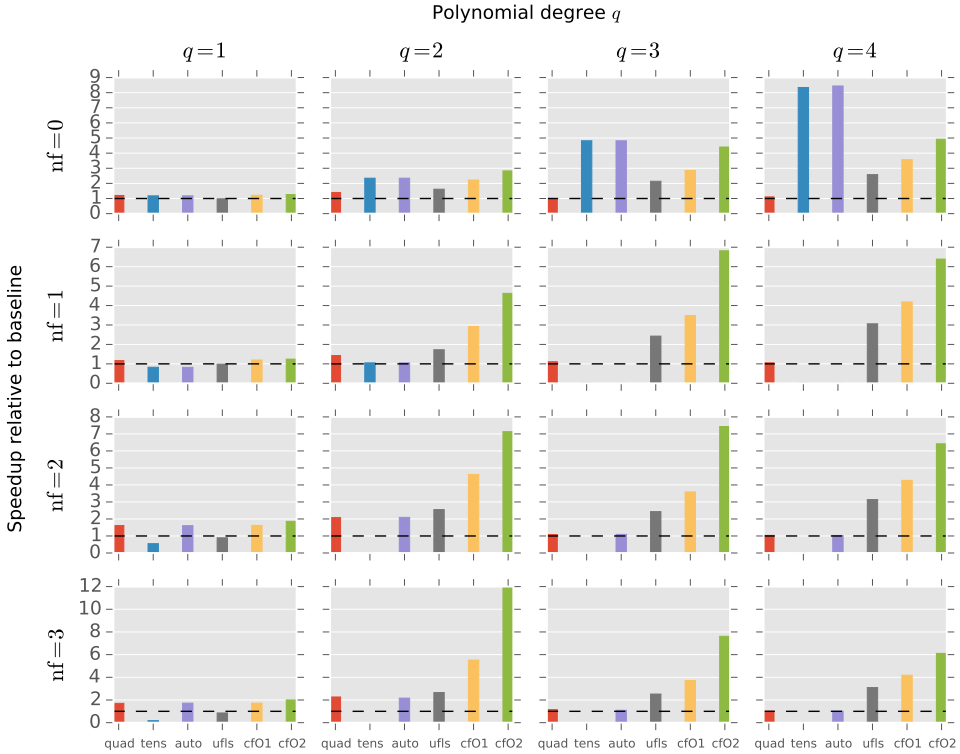

Fig. 9: Performance evaluation for the bilinear form arising in an *elastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

global matrix in compressed sparse row format. This unsurprisingly leads to a very low arithmetic intensity (i.e., $\frac{\text{operation count}}{\text{bytes moved}}$), which collocates the kernels in a memory-bound regime.

For $q \geq 2$, instead, the transformation systems impact the performance, and `cf02` generally shows the highest speed-ups. It is worth noting that `auto` does not always select the fastest implementation, as it always opts for `tens`, while for $\text{nf} \geq 2$ quad tends to be preferable. On the other hand, `cf02` always makes the optimal decision about whether to apply pre-evaluation or not. Unexpectedly, in spite of the simplicity of the form, the performance of the various code generation systems can differ significantly.

Helmholtz. As in the case of mass matrix, when $q = 1$ the matrix insertion phase is dominant. For $q \geq 2$, the general trend is that `cf02` outperforms the competitors. In particular:

- $\text{nf} = 0$. pre-evaluation makes `cf02` notably faster than `cf01`, especially for high values of q; `auto` correctly selects `tens`, which is comparable to `cf02`.
- $\text{nf} = 1$. `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 `cf02` and `tens`/`auto`: `cf02` applies sharing elimination, but it correctly avoids pre-evaluation because of the excessive expansion cost.

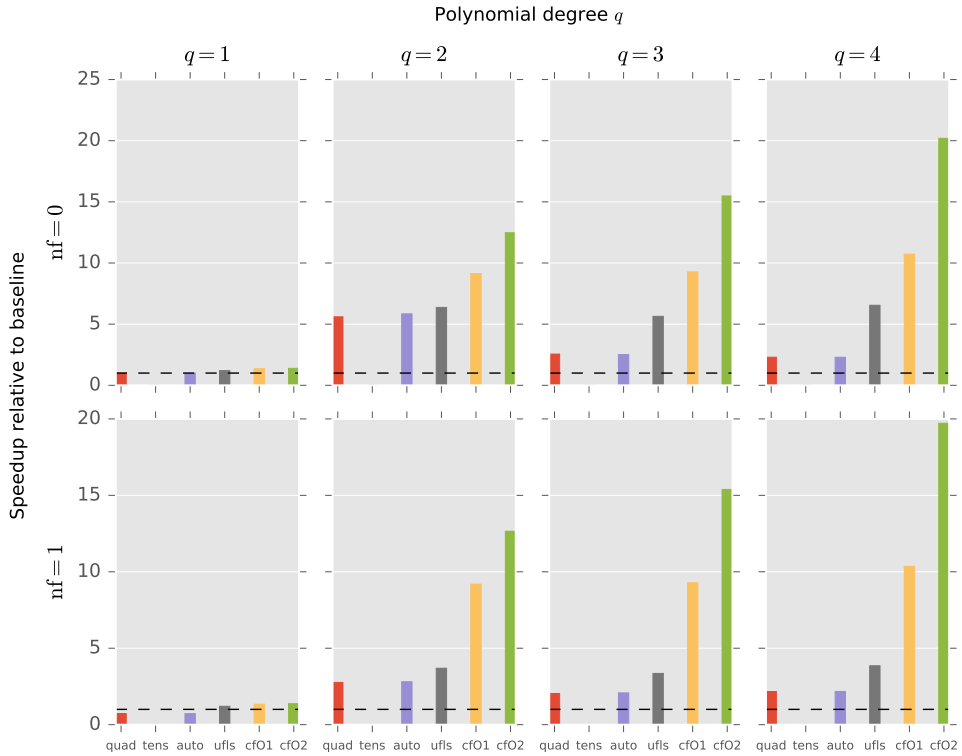

Fig. 10: Performance evaluation for the bilinear form arising in a *hyperelastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

$\text{nf} = 2$ *and* $\text{nf} = 3$. `auto` reverts to quad, which would theoretically be the right choice (the flop count is much lower than in `tens`); however, the generated code suffers from the presence of indirection arrays, which break autovectorization and "traditional" code motion.

In the case $\langle \text{nf} = 0,\ q = 2 \rangle$, the $2\times$ speed-ups achieved by `cf02` and `tens` are due to the remarkable drop in operation count achieved by pre-evaluating reductions: from order 10^4 (`ufls` $3 \cdot 10^4$, `cf01` $2 \cdot 10^4$) to order 10^3. By unrolling loops and eliminating operations over zeros, `tens` eventually executes fewer operations than `cf02` (roughly, 10^3 instead of $2 \cdot 10^3$), but the performance is basically identical because of matrix insertion, which plays again the limiting role. With 10^2 entries to be updated in the global matrix, following the same procedure of the previous paragraph, it is straightforward to verify that $\frac{\text{operation count}}{\text{bytes moved}} \ll 1$, a typical symptom of memory-boundedness.

Another relevant problem instance is $\langle \text{nf} = 0,\ q = 4 \rangle$, in which `cf02` performs $1.5\times$ more operations than `tens`, but still exhibits a better execution time. This is again caused by the chosen code generation strategies: while `tens` prefers to fully unroll the computation to detect all avoidable operations, `cf02` preserves the loop structure. This choice mitigates the memory transfer (especially between L1 cache and registers) by maximizing register locality.

The slow-downs (or marginal improvements) seen in a small number of cases exhibited by `ufls` can be attributed to the presence of sharing in the generated code. Further, the factorization applied by `ufls` hides code motion opportunities that the underlying compiler would otherwise have been able to exploit.

An interesting experiment we additionally performed was relaxing the memory threshold by setting $T_H = \text{size(L3)}$. We found that this makes `cf02` generally slower for $\text{nf} \geq 2$, with a maximum slow-down of $2.16\times$ with $\langle \text{nf} = 2, q = 2 \rangle$. This effect could be worse when running in parallel, since the L3 cache is shared and different threads would end up competing for the same resource.

Elasticity. The results for the elastic model are displayed in Figure 9. The main observation is that `cf02` never triggers pre-evaluation, although in some occasions it should. To clarify this, consider the test case $\langle \text{nf} = 0,\ q = 4 \rangle$, in which `tens/auto` show a considerable speed-up over `cf02`. Although pre-evaluation is determined profitable by `cf02` in terms of operation count, it is eventually not applied to avoid exceeding T_H. However, running the same experiments with $T_H = \text{size(L3)}$ resulted in a dramatic improvement, even higher than that obtained by `tens`. The reason is that, despite exceeding T_H by roughly 40%, the saving in operation count is so large ($5\times$ in this specific problem) that pre-evaluation would in practice be the winning choice. This suggests that our objective function should be improved to handle the cases in which there is a significant gap between potential cache misses and reduction in operation count.

We also note that:

— the differences between `cf02` and `cf01` are due to the perfect sharing elimination and the zero-valued blocks avoidance technique presented in Section 6.3.
— when $\text{nf} = 1$, `auto` prefers `tens` over `quad`, which leads to sub-optimal operation counts and execution times.
— `ufls` often results in better execution times than `quad` and `tens`. This is due to multiple factors, including avoidance of indirection arrays, preservation of loop structure, and a more effective code motion strategy.

We now consider the test case $\langle \text{nf} = 2,\ q = 3 \rangle$ to discuss the relationship between achieved performance and operation count, although analogous considerations apply to the other scenarios. It is difficult to estimate the number of flops executed by the code generated by the FEniCS Form Compiler, since loop-invariant code motion is extensively applied by the underlying compiler. The calculation is instead simple in all other cases: while `cf02` performs roughly $2 \cdot 10^5$ operations, `quad`, `ufls` and `cf01` are much closer to 10^6 operations. In particular, `ufls` requires $4.3\times$ more operations than `cf02`, whereas the execution time speed-up is only slightly bigger than $2\times$. The distance between these two values may be attributed to the memory transfer cost – especially inserting the $3.6 \cdot 10^3$ non-zero entries of the local element matrix – which, for reasons already discussed, becomes a bottleneck. `quad`'s operation count is even lower than that of `ufls` ($84 \cdot 10^4$ instead of $97 \cdot 10^5$), but as explained earlier, the use of indirection arrays dramatically affects the low level efficiency. Despite requiring roughly 10^5 fewer operations than `cf02`, `tens` is again affected by the poor low level efficiency of the generated code.

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 result, since it indicates that our transformation algorithm scales well with form complexity. The fact that all code transformation systems (apart from `tens`) show quite significant speed-ups suggests two points. First, the baseline is highly inefficient. With forms as complex as in the hyperelastic model, a trivial translation

of integration routines into code should always be avoided as even the best general-purpose compiler available (the Intel compiler on an Intel platform at maximum optimization level) fails to exploit the structure inherent in the expressions. Second, the strategy for removing spatial and temporal sharing has a tremendous impact. Sharing elimination as performed by `cf02` ensures a critical reduction in operation count, which becomes particularly pronounced for higher values of q.

We also tried to relate the achieved performance to the operation counts obtained by each transformation system. We here limit ourselves to consider the case $\langle \mathrm{nf} = 0,\ q = 4 \rangle$, for which the speed-ups are generally massive. For the original code, at the source level $37 \cdot 10^9$ operations are counted. However, this number is in practice much lower, since by inspection of the assembly code generated by the underlying compiler, it is clear that common sub-expressions elimination and loop-invariant code motion have been applied in aggressive way. The actual operation count, however, is definitely larger than the 10^8 operations performed by `ufls`. `cf02` brings the operation count down to $3 \cdot 10^7$; that is, more than $3\times$ lower than `ufls`. This is completely in line with the drop in execution times, with `cf02` outperforming `ufls` by slightly more than $3\times$.

Unsurprisingly, the memory transfer impact is less relevant in this hyperelastic than in all previous forms. With $\langle \mathrm{nf} = 0,\ q = 1 \rangle$, `cf02` is already $1.4\times$ faster than the untransformed code, and the gain increases with both nf and q.

8. CONCLUSIONS

We have developed a theory for the optimization of finite element integration loop nests. The article details the domain properties which are exploited by our approach (e.g., linearity) and how these translate to transformations at the level of loop nests. All of the algorithms shown in this paper have been implemented in COFFEE, a compiler publicly available fully integrated with the Firedrake framework. The correctness of the transformation algorithm was discussed. The performance results achieved suggest the effectiveness of our methodology.

9. LIMITATIONS AND FUTURE WORK

We have defined sharing elimination and pre-evaluation as high level transformations on top of a specific set of rewrite operators, such as code motion and factorization, and we have used them to construct the transformation space. There are three main limitations in this process. First, we do not have a systematic strategy to optimize sub-expressions which are independent of linear loops. Although we have a mechanism to determine how much computation should be hoisted to the level of the integration (reduction) loop, it is not clear how to effectively improve the heuristics used at step (6) in Algorithm 1. Second, lower operation counts may be found by exploiting domain-specific properties, such as redundancies in basis functions; this aspect is completely neglected in this article. Third, with Constraint 1 we have limited the applicability of code motion. This constraint was essential given the complexity of the problem tackled.

Another issue raised by the experimentation concerns selecting a proper threshold for Constraint 2. To solve this problem would require a more sophisticated cost model, which is an interesting question deserving further research.

We also identify two additional possible research directions: a complete classification of forms for which a global optimum is achieved; and a generalization of the methodology to other classes of loop nests, for instance those arising in spectral element methods.

REFERENCES

Martin Sandve Alnæs. 2016. UFLACS - UFL Analyser and Compiler System. https://bitbucket.org/fenics-project/uflacs. (2016).

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

Martin Sandve Alnæs, Anders Logg, Garth Wells, Lawrence Mitchell, Marie E. Rognes, Miklós Homolya, Aslak Bergersen, David A. Ham, Johannes Ring, chrisrichardson, and Kent-Andre Mardal. 2016. ufl: The Unified Form Language. (April 2016). DOI:http://dx.doi.org/10.5281/zenodo.49282

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. 2016. petsc4py: The Python interface to PETSc. (April 2016). DOI:http://dx.doi.org/10.5281/zenodo.49283

Miklós Homolya and Lawrence Mitchell. 2016. TSFC - Two-stage form compiler. https://github.com/firedrakeproject/tsfc. (2016).

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, Martin Sandve Alnæs, Marie E. Rognes, Garth Wells, Johannes Ring, Lawrence Mitchell, Johan Hake, Miklós Homolya, Florian Rathgeber, Fabio Luporini, Graham Markall, Aslak Bergersen, Lizao Li, David A. Ham, Kent-Andre Mardal, Jan Blechta, Gheorghe-Teodor Bercea, Tuomas Airaksinen, Nico Schlömer, Hans Petter Langtangen, Colin J Cotter, Ola Skavhaug, Thomas Hisch, mliertzer, Joachim B Haga, and Andrew T. T. McRae. 2016. ffc: The FEniCS Form Compiler. (April 2016). DOI:http://dx.doi.org/10.5281/zenodo.49276

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, Lawrence Mitchell, Miklós Homolya, Florian Rathgeber, David A. Ham, Michael Lange, Graham Markall, and Francis Russell. 2016. COFFEE: A Compiler for Fast Expression Evaluation. (April 2016). DOI:http://dx.doi.org/10.5281/zenodo.49279

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

Lawrence Mitchell, David A. Ham, Florian Rathgeber, Miklós Homolya, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Michael Lange, Colin J Cotter, Christian T. Jacobs, Fabio Luporini, Simon Wolfgang Funke, Henrik Büsing, Tuomas Kärnä, Anna Kalogirou, Hannah Rittich, Eike Hermann Mueller, Stephan Kramer, Graham Markall, Patrick E. Farrell, Geordie McBain, and Asbjørn Nilsen Riseth. 2016. firedrake: an automated finite element system. (April 2016). DOI:http://dx.doi.org/10.5281/zenodo.49284

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

Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2015. Firedrake: automating the finite element method by composing abstractions. *CoRR* abs/1501.01809 (2015). http://arxiv.org/abs/1501.01809

Florian Rathgeber, Fabio Luporini, and Lawrence Mitchell. 2016a. firedrake-bench: firedrake bench optimality paper release. (April 2016). DOI:http://dx.doi.org/10.5281/zenodo.49290

Florian Rathgeber, Lawrence Mitchell, Fabio Luporini, Graham Markall, David A. Ham, Gheorghe-Teodor Bercea, Miklós Homolya, Andrew T. T. McRae, Hector Dearman, Christian T. Jacobs, gbts, Simon Wolfgang Funke, Kaho Sato, and Francis Russell. 2016b. PyOP2: Framework for performance-portable parallel computations on unstructured meshes. (April 2016). DOI:http://dx.doi.org/10.5281/zenodo.49281

Marie E. Rognes, Anders Logg, David A. Ham, Miklós Homolya, Nico Schlömer, Jan Blechta, Andrew T. T. McRae, Aslak Bergersen, Colin J Cotter, Johannes Ring, and Lawrence Mitchell. 2016. fiat: The Finite Element Automated Tabulator. (April 2016). DOI:http://dx.doi.org/10.5281/zenodo.49280

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).

Barry Smith, Satish Balay, Matthew Knepley, Jed Brown, Lois Curfman McInnes, Hong Zhang, Peter Brune, sarich, stefanozampini, Dmitry Karpeyev, Lisandro Dalcin, tisaac, markadams, Victor Minden, Victor-Eijkhout, vijaysm, Karl Rupp, Fande Kong, and SurtaiHan. 2016. petsc: Portable, Extensible Toolkit for Scientific Computation. (April 2016). DOI:http://dx.doi.org/10.5281/zenodo.49285