

On Optimality of 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 and quasi-optimality are defined in terms of floating point operations given a memory bound. We provide an approach for exploring the space of legal transformations, which systematically reaches quasi-optimality. The conditions under which optimality is achieved are also discussed. A theoretical analysis and extensive experimentation, which shows significant performance improvements over several state-of-the-art 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. On Optimality of Finite Element Integration. *ACM Trans. Arch. & Code Opt.* V, N, Article A (January YYYY), 24 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 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;

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 1539-9087/YYYY/01-ARTA \$15.00

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

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 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 *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 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. 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 7 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 that solves this problem for a particular class of forms and provides very good approximations in all other cases (we will define “quasi-optimality”, which relates operation count with inner loops). Summarizing, our contributions are as follows:

- We formalize 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 algorithm systematically leads to quasi-optimality. We also explain under what conditions of the input problem optimality is achieved.
- We integrate our approach with a compiler, COFFEE¹, 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).

¹COFFEE 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:

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

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

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

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

2.1. Monomials

It has been shown (e.g., 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 $F_K : K_0 \rightarrow K$ 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

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

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.

2.3. Tensor contraction mode

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

$$A_{ij}^K = \sum_{\alpha_1=1}^d \sum_{\alpha_2=1}^d \sum_{\alpha_3=1}^n \det F'_K w_{\alpha_3} \sum_{\beta=1}^d \frac{X_{\alpha_1}}{\partial x_{\beta}} \frac{\partial X_{\alpha_2}}{\partial x_{\beta}} \int_{K_0} \phi_{\alpha_3} \frac{\partial \phi_i}{\partial X_{\alpha_1}} \frac{\partial \phi_j}{\partial X_{\alpha_2}} dX. \quad (8)$$

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

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

in which the pre-evaluated “reference tensor” $A_{i_1 i_2 \alpha}^0$ 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 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 7). 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 and quasi-optimality for finite element integration as well as the space of legal transformations that needs be explored to achieve them. How the 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 .

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 synthesize quasi-optimal code.

We define two other classes of loops.

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 (right hand side of a statement) result from the translation of a form in high level matrix notation

```

for (e = 0; e < E; e++)
  ...
  for (i = 0; i < I; i++)
    ...
    for (j = 0; j < J; j++)
      for (k = 0; k < K; k++)
        
$$a_{ejk} += \sum_{w=1}^m \alpha_{eij}^w \beta_{eik}^w \sigma_{ei}^w$$


```

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

into code. In particular, m is the number of “terms” (a monomial can be implemented by one or more terms), $\alpha_{eij} (\beta_{eik})$ represents the product of a coefficient function (e.g., the inverse Jacobian matrix for the change of coordinates) with test (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), coefficients (e.g., 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 (e.g., products have not been expanded yet, distinct monomials can still be identified, etc.). 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 $\Gamma : \Lambda \rightarrow \Lambda'$ 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 total number of operations (additions, products) performed to evaluate the result is minimal.*

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

Definition 9 (Flop-decreasing transformation). *A sequence of transformations reducing the operation count is called flop-decreasing.*

Definition 10 (Quasi-optimality of a loop nest). *Given Λ , Λ' and Γ as in Definition 8, we say that $\Lambda' = \Gamma(\Lambda)$ is a quasi-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 “deceptive” solutions 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 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 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. On the other hand, multilinearity suggests a certain degree of regularity for α_{eij} and β_{eik} . In the following sections, we will elaborate on these observations and formulate an approach that achieves: (i) at least quasi-optimality in all cases; (ii) 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 – 3.5)
- 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 $\{L_{i_1}, L_{i_1}, \dots, L_{i_n}\} \subset \Lambda$.

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 seeks sub-expressions that are redundantly executed by at least one loop in the nest², leads to optimality (Figure 2(c)).

<pre> for (j = 0; j < J; j++) for (i = 0; i < I; i++) a_{ji} += b_jc_i + b_jd_i </pre>	<pre> for (j = 0; j < J; j++) for (i = 0; i < I; i++) a_{ji} += b_j(c_i + d_i) </pre>	<pre> 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_jt_i </pre>
(a) With spatial sharing	(b) With temporal sharing	(c) Optimal form

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

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 quasi-optimality³, with optimality

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

³This requires coordination with other transformations, as discussed in the following sections. For the moment – and for ease of exposition – we neglect this aspect.

being achieved if stronger preconditions hold. Setting quasi-optimality, instead of 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 $J^{-1}\nabla v \cdot J^{-1}\nabla v$, with J^{-1} being the inverse Jacobian matrix for the change of coordinates and v a generic two-dimensional vector. The tensor operation will reduce to the scalar expression $(av_i^0 + bv_i^1)(av_i^0 + bv_i^1) + \dots$, 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 $v_i^0 v_i^0 aa + v_i^0 v_i^1 (ab + ab) + v_i^1 v_i^1 cc + \dots$ 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(s) size, the increase in operation count due to expansion (in Strategy 2), and the gain in 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 in order to eliminate the spatial sharing induced by any s_Λ – a generic symbol depending on at least one loop in Λ – it is sufficient to factorize all occurrences of s_Λ in the expression.*

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

Proof. This descends 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. \square

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, in which the factorization of s_Λ only “minimizes” (rather than “eliminates”) spatial sharing, applies (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).

- (1) Perform a depth-first visit of the expression tree to collect and partition multilinear operands into disjoint sets, $\mathbb{P} = \{P^1, \dots, P^p\}$. \mathbb{P} is such that all multilinear operands in 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 $|P| < |S_P|$, apply Strategy 1.
Note: $|P|$ and $|S_P|$ represent the number of products in the innermost loop induced by P if, respectively, Strategy 1 or Strategy 2 were applied. This is a consequence of Proposition 1.
- (2) Build the *sharing graph* $G = (S, E)$. Each $s \in S$ represents a multilinear symbol or a temporary produced at step (2). 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.
- (3) Let $T \subset S$ be the subset of vertices with at most one incident edges, or “terminals”. Partition T into disjoint sets, $\mathbb{T} = \{T^1, \dots, T^t\}$, such that the terminals in T^i are either adjacent to the same vertex or disconnected. Apply Strategy 2 factorizing the terminals in each T^i , provided that this leads to a reduction in operation count.
Note: the last check ensures the flop-decreasing nature of the transformation; the cases in which the expansion cost exceeds the code motion gain are discarded.
- (4) Partition S into disjoint sets, $\mathbb{S} = \{S^1, \dots, S^n\}$, such that S^i includes all instances of a given symbol s in the expression. Transform G by merging $\{s^1, \dots, s^m\} \subset S^i$ into a unique vertex s (taking the union of the edges), provided that factorizing $[s^1, \dots, s^m]$ would not cause an increase in operation count.
Note: the variation in operation count is a function of (number and arity) of the products wrapping the elements in S^i , which impact the expansion cost.
- (5) Restructure the expression based on the structure of G and a set of rewrite rules *<precondition: action>*, to be applied in the order given below.
 - (A) $\{t^1, \dots, t^n\} \subset T$ adjacent to $s \in S$, $n > 1$: take s and apply Strategy 2.
Note: this recasts the expression as $s(\dots t^1 + \dots + \dots t^n)$.
 - (B) Only one terminal t adjacent to s : take an arbitrary non-terminal symbol s^{nt} adjacent to s and apply Strategy 2.
Note: this recasts the expression as $s(\dots t + \dots s^{nt})$.
 - (C) $T = \emptyset$, $C = [s, \dots, v, \dots, s]$ is a cycle, v has highest degree in C , $\{s^1, \dots, s^n\}$ adjacent to v : take v and apply Strategy 2.
Note: this recasts the expression as $v(\dots s^1 + \dots + \dots s^n)$.
 Rule (C) may expose further hits for rules (A) and (B). Rules are therefore applied cyclically until no more hits are possible.
Note: the last two steps only impact bilinear forms, since otherwise $E = \emptyset$.
- (6) Perform a depth-first visit of the expression 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.

Remark. Although the primary goal of Algorithm 1 is operation count minimization within the multilinear loop nest, the enforcement of flop-decreasing transformations (steps (3) 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 $T = \{c_i, d_i\}$. Rewrite rule (4A) is hit, which leads to the code in Figure 2(c).

In Figure 3, Algorithm 1 is executed in a 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)) $G = (\{t^0, t^1, t^2, t^3\}, \{(t^0, t^2), (t^1, t^3)\})$, although no rewrite rule is applicable since $S = T$

<pre> 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} += (((z⁰a_{ik} + z²b_{ik})* (z⁰c_{ij} + z²d_{ij}))+ ((z¹a_{ik} + z³b_{ik})* (z¹c_{ij} + z³d_{ij}))) * W_i * det </pre> <p style="text-align: center;">(a) Normal form</p>	<pre> for (e = 0; e < E; e++) ... for (i = 0; i < I; i++) for (k = 0; k < K; k++) t_k⁰ = (z⁰a_{ik} + z²b_{ik}) t_k¹ = (z¹a_{ik} + z³b_{ik}) for (j = 0; j < J; j++) t_j² = (z⁰c_{ij} + z²d_{ij}) * W_i * det t_j³ = (z¹c_{ij} + z³d_{ij}) * W_i * det for (j = 0; j < J; j++) for (k = 0; k < K; k++) a_{ejk} += t_k⁰ * t_j² + t_k¹ * t_j³ </pre> <p style="text-align: center;">(b) After sharing elimination</p>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 3: Applying sharing elimination to the bilinear form arising from a Poisson equation in 2D. The operation counts are $E(f(z^0, z^1, \dots) + IJK \cdot 18)$ (left) and $E(f(z^0, z^1, \dots) + I(J \cdot 6 + K \cdot 9 + JK \cdot 4))$ (right), with $f(z^0, z^1, \dots)$ representing the operation count for evaluating z^0, z^1, \dots , and common sub-expressions being counted once. Note that the synthesis in Figure 3(b) reaches optimality apart from the pathological case $I, J, K = 1$.

A much more complex example, using the hyperelastic model evaluated in Section 7, is made available online⁴.

3.3. Monomial pre-evaluation

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 ex-

⁴Sharing elimination example - application to a hyperelastic model: <https://gist.github.com/FabioLuporini/14e79457d6b15823c1cd>

emplified 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).

<pre> for (e = 0; e < E; e++) for (i = 0; i < I; i++) for (k = 0; k < K; k++) a_{ek} += d_eb_{ik}c_i + d_eb_{ik}d_i </pre>	<pre> 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_et_k </pre>
(a) With reduction	(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” tensors. 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 feasible⁵, 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 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 $\langle t, n, S \rangle$, pre-evaluation will shift the loop nest from a CPU-bound to a memory-bound regime, 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.

⁵For 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>

3.4. Memory constraints

We have just observed that the code motion induced by monomial pre-evaluation may excessively increase the working set size. Interestingly, 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.

3.5. Limitation in the optimization space considered

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. We remark that sharing elimination and pre-evaluation aim to minimize spatial and temporal redundancy relying on *a specific set of operators*. Theoretically, lower operation counts may be found by exploiting other domain-specific properties, such as redundancies in basis functions. Constraint 1 also limits the applicability of code motion, which further reduced the transformation space to be analyzed.

4. SELECTION AND COMPOSITION OF TRANSFORMATIONS

In this section, we build a transformation algorithm that, given a memory bound, systematically produces quasi-optimal 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 with respect to 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 (e.g. 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 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 $\theta : M \rightarrow \mathbb{Z}$ 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: 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 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} 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 bipartitions analyzed by Algorithm 3. We therefore seek an analytic way of determining θ^{pre} .

```

// 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++)
            aejk += 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++)
                aejk += H(...)

```

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

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. 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 $f_0 * b_{i0} + f_1 * b_{i1} + f_2 * b_{i2}$ represents the coefficient f over (tabulated) basis functions (array B). In order to apply pre-evaluation, the expression needs to 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++)
            aijk += bij * bik * (f0 * Bi0 + f1 * Bi1 + f2 * Bi2)

```

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., $b_{i0} * b_{i1} * \dots$ and $b_{i1} * b_{i0} * \dots$) 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 as performed by the transformation algorithm guarantees quasi-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 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 quasi-optimal 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 produces quasi-optimal (or optimal) loop nests.*

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 for determining an optimum preserving Constraint 2⁶. 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 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

⁶Note that the problem can be seen as an instance of the well-known Knapsack problem

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, a set of rewrite rules are applied. We here discuss why applying, in rigorous order, rules (A), (B), and (C), leads to the optimum.

Rules (A) and (B) represent “unavoidable” factorizations, for which one multiplication must be paid no matter how Strategy 2 is scheduled. In particular:

Rule (A). The terminals $\{t^1, \dots, t^n\}$ can only be grouped through the factorization of s (by definition of terminal), so this rule has no side effects.

Rule (B). Since s^{nt} is not a terminal, there must exist at least one edge between s^{nt} and a generic vertex v . Let us consider the alternative factorization $s^{nt}(\dots s + \dots + \dots v)$. If v were a terminal, we observe that setting $v = t$ and $s = s^{nt}$ would simply expose another hit for rule (B). Otherwise (v non terminal), we would have s now appearing in two different sub-expressions $((\dots s + \dots + \dots v)$ and $(\dots s t)$), which would clearly be sub-optimal as a consequence of the spatial sharing induced by s . From this, we evince that applying (C) prior (B) may prune the optimum from the search space.

To reach the optimum, rule (C) discards all those potential factorizations that can be rather “aggregated” with *essential* factorizations. A factorization is essential if a vertex is in a cycle. Being in a cycle is the fundamental condition to

Having higher degree simply means grouping more symbols (i.e., avoiding more products);

□

Throughout the paper we have reiterated the idea that Algorithm 3 reaches optimality if stronger preconditions on the input variational form are satisfied. We here discuss such preconditions, in increasing order of complexity.

- (1) There is a single monomial and only a specific 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 would avoid ambiguity in factorization, which in turn would guarantee the output of step (7) in Algorithm 1 be optimality.
- (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” operators, 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: COFFEE can be seen as a lightweight, low level computer algebra system, not necessarily tied to finite element integration. Thirdly, robustness: the same operators are exploited, and therefore stressed, by different optimization pipelines. The rewrite operators, whose (Python) implementation is based on manipulation of abstract syntax trees (ASTs), compose 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). In Firedrake, for example, a modified version of the FEniCS Form Compiler producing ASTs, instead of strings, is used. Thus, COFFEE decouples the mathematical manipulation of a form from code optimization; or, in other words, relieves form compiler developers of caring about performance 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, promotes 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 a way 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 for determining which zero-valued blocks 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`.

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 available⁷.

⁷https://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 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” (no code generation time, no compilation time). They include 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 (publicly available) were written for the compatibility of quad, tens, and ufls 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 $T_H = L2_{size}$; 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 $T_H = L3_{size}$. 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 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)

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 $\langle form, nf, q \rangle$, 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.



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

The rest of the section is organized as follows: we first provide insights into the overall outcome of the experimentation; we then comment on the impact of a fundamental low-level optimization, namely autovectorization; finally, we explain, in detail and 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 impact 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 adopted when exceeding T_H . We will later thoroughly elaborate on all these aspects.



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.

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, cf01, and cf02 the iteration spaces have similar structure (there are a few exceptions in cf02 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.

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 $q \geq 2$, cf02 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

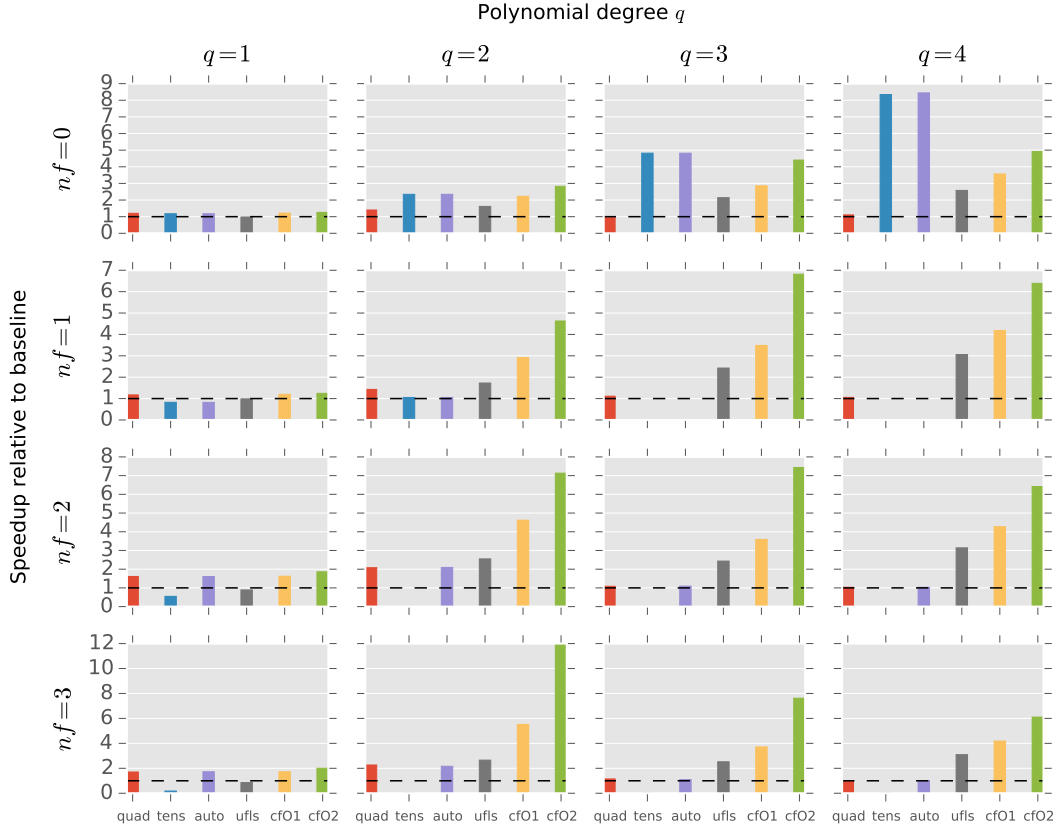


Fig. 9: Performance evaluation for the bilinear form arising in an *elastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

to be preferable. On the other hand, cf02 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 cf02 outperforms the competitors. In particular, with

- $nf = 0$, the adoption of pre-evaluation by cf02 results in increasingly notable speed-ups over cf01, as q increases; tens is comparable to cf02, with auto making the right choice.
- $nf = 1$, 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.



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.

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 $2.16\times$ 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.

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 $\langle nf = 0, q = 2 \rangle$, in which tens/auto show a considerable speed-up over cf02. cf02 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 cf02 and cf01 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, 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. Firstly, 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 (Intel's on an Intel platform at maximum optimization level) cannot exploit the structure inherent in the expressions. Secondly, 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 .

8. CONCLUSIONS

We have developed transformation algorithms and a theory of optimality for finite element integration. The implementation was carried out in COFFEE, a compiler fully integrated with the Firedrake framework, and therefore made available to an increasingly significant user base. Performance results show the effectiveness of our methodology. There are still several open problems. The main one is an elaborate classification of forms for which optimality can be reached. Secondly, we would like to extend our methodology to classes of loops arising in spectral methods, in which interaction with low level optimization becomes even more critical due to the larger working sets.

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>
- 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 Fi-

- nite 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>
- 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>
- 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).