A

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 to explore the space of legal transformations and discuss under what conditions optimality or quasi-optimality hold. A theoretical analysis and extensive experimentation, which shows systematic 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), 23 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;

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 is Section 6). Research compilers, for instance those based on polyhedral analysis of loop nests such as PLUTO ([Bondhugula et al. 2008]), focus on parallelization and optimization for cache locality, treating issues orthogonal to the question of minimising flops. The lack of suitable third-party tools has led to the development of a number of domain-specific code transformation (or synthesizer) systems. In Ølgaard and Wells [2010], it is shown how automated code generation can be leveraged to introduce optimizations that a user should not be expected to write "by hand". In Kirby and Logg [2006] and Russell and Kelly [2013], mathematical reformulations of finite element integration are studied with the aim of minimizing the operation count. In Luporini et al. [2015], the effects and the interplay of generalized code motion and a set of low level optimizations are analysed. It is also worth mentioning an on-going effort to produce a new form compiler, called UFLACS ([Alnæs 2015]), which adds to the already abundant set of code transformation systems for assembly operators. The performance evaluation in Section 6 includes most of these optimization systems.

However, in spite of such a considerable research effort, still there is no answer to one fundamental question: can we automatically generate an implementation of a form which is optimal in the number of flops executed? In this paper, we formulate an approach that solves this problem for a particular class of forms and provides very good approximations ("quasi-optimality") in all other cases. 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 explain under what conditions our algorithm leads to optimality. In particular, we show that (i) for a particular class of problems, optimality is reached; (ii) quasi-optimality is, in general, always achieved (i.e., the operation count is at least optimal in innermost loops).
— 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 an engine based on symbolic execution that allows skipping irrelevant floating point operations (e.g., those involving zero-valued quantities).

2. PRELIMINARIES

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

We consider the weak formulation of a linear variational problem

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

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

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

in which A and b discretize a and L respectively:

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

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

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

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

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

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

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

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

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

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

2.1. Monomials

In general, 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

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

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" $A_{i_1 i_2 \alpha}$ 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 6). 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 we need to explore to achieve it. 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 linear 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 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 (Free order loop)**.** *A loop L_i is said to be a free order loop if its iterations can be executed in any arbitrary order; that is, there are no loop-carried dependencies across different iterations.*

To contextualize, consider Equation 7 and the (abstract) loop nest implementing it illustrated in Figure 1. The imperfect nest $\Lambda = [L_e, L_i, L_j, L_k]$ comprises a free order loop L_e (over 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

```
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.

(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 "terms" (a monomial can be implemented by one or more terms), α_{eij} (β_{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 particular 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 bring us to formalize the class of loop nests for which we seek optimality.

Definition 7 (Finite element integration loop nest)**.** *A finite element integration loop nest is a loop nest in which we identify, in order, an imperfect free order loop, a (generally) imperfect, linear or non-linear reduction loop, and a multilinear loop nest whose body is an independent basic block in which each statement has expressions in normal form.*

For a finite element integration loop nest, we characterize optimality and quasi-optimality as follows.

Definition 8 (Optimality of a loop nest)**.** *Let Λ be a generic loop nest, and let Γ be a generic transformation function $\Gamma : \Lambda \to \Lambda'$ such that Λ' is semantically equivalent to Λ (possibly, $\Lambda' = \Lambda$). We say that $\Lambda' = \Gamma(\Lambda)$ is an optimal synthesis of Λ if the number of operations (additions, products) that it performs to evaluate the result is minimal.*

Definition 9 (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) that it performs to evaluate the result is minimal in all innermost loops.*

Note that Definitions 8 and 9 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 element, this is often true for more complex multilinear forms and/or higher order elements. We also note that quasi-optimality approximates optimality very well whenever the inner loop trip counts are "sufficiently large".

Achieving optimality in polynomial time is not generally feasible, since the σ_{ei} sub-expressions can be arbitrarily unstructured. On the other hand, multilinearity ensures a certain degree of regularity for the α_{eij} and β_{eik} sub-expressions. In the following sections, we will elaborate on these observations and formulate an approach that achieves: (i) optimality whenever the σ_{ei} sub-expressions are "sufficiently regular"; (ii) quasi-optimality in all other cases. 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 10 (Sharing)**.** *A statement within a loop nest Λ presents sharing if at least one of the following conditions hold:*

(*1*) *there are at least two symbolically identical sub-expressions (spatial sharing)*
(*2*) *there is at least one non-trivial sub-expression (an addition or a product) that is redundantly executed as independent of $\{L_{i_0}, L_{i_1}, ...L_{i_n}\} \subset \Lambda$ (temporal sharing)*

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 b_j. Factorization eliminates spatial sharing and promotes temporal sharing (Figure 2(b)). Finally, generalized code motion [Luporini et al. 2015] 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.

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 expressions were systematically unstructured, 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, attempts to exploit the structure inherent in finite element integration expressions to guarantee quasi-optimality[2], with optimality being achieved if stronger preconditions hold. Relaxing the problem is essential to produce simple and computationally efficient algorithms – two necessary conditions for integration with a compiler.

Section 3.2.1 discusses structural and algebraic properties characterizing our expressions. Section 3.2.2 presents the sharing elimination algorithm. Examples are provided in Section 3.2.3. We recall that we are assuming a generic finite element integration loop nest $\Lambda = [L_e, L_i, L_j, L_k]$ with expressions in normal form.

3.2.1. Structured tensor operations. Finite element expressions can be seen as composition of operations between tensors. We observe that, when multiplying two tensors, it often happens that the optimal scheduling strategy is to be searched in a space of size 2. To illustrate the concept, we take a matrix-vector product $y_{\Psi\Delta} = A_{\Psi} x_{\Delta}$, in which the entries of y, A and x depend on $\Psi, \Delta \subset \Lambda$. Let us assume that $y_{\Psi\Delta}$ itself is an operand of a larger scalar-valued expression, so its entries can represent any possible sub-expression in $\{\alpha_{eij}, \beta_{eik}, \sigma_{ei}\}$. For example, suppose that A is the inverse

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

Jacobian matrix of a given coefficient and x the gradient of a function in two dimensions, and consider the case $\Psi = \emptyset$, $\Delta = \{L_i\}$. Thus an entry in y_{L_i} is of the form $(af_i + bg_i)(cf_i + dg_i) \in \sigma_{ei}$. To schedule this expression, we basically have two options:

(i) Applying generalized code motion – not needed in the example as we have one loop.
(ii) Searching for spatial sharing, expanding and factorizing the expression to promote temporal sharing – the expression in the example would be recast as $f_i(a+c)+g_i(b+d)$, with $(a + c)$ and $(b + d)$ being two hoistable sub-expressions.

The reader can verify that (ii) improves the operation count over (i) for any size of L_i. In general, however, the optimal option depends on multiple factors: the loop size, the expansion cost, and code motion. One can appreciate the dichotomy between (i) and (ii) in several expressions originating from a wide range of variational forms[3]. We then regard as *structured tensor operation* any tensor operation that result in sub-expressions for which there is no ambiguity in spatial sharing elimination – implying that the optimal operation scheduling is to be determined out of two alternatives.

Of notable importance, from the perspective of achieving quasi-optimality, is the fact that sub-expressions depending on the multilinear loop nest always result from structured tensor operations. Often, structured tensor operations characterize the σ_{ei} sub-expressions too. For instance, they systematically arise in the weak variational form of the complex hyperelastic model analyzed in Section 6 (e.g., the Cauchy-Green tensor).

3.2.2. Algorithm. Algorithm 1 describes sharing elimination assuming as input a tree representation of the loop nest. The algorithm is meant to be applied to each statement appearing in the independent basic block of a finite element integration loop nest. It exploits the observation that test and trial functions are always part of structured tensor operations, due to the nature of finite element integration. Section 4.3 will elucidate the claims of quasi-optimality and optimality.

The algorithm makes use of the following notation and terminology:

— $|.|$: a generic "sizeof" operator (e.g., cardinality of a set, extent of a loop).
— *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.).
— *strategy (i)* and *strategy (ii)*: the two approaches described in Section 3.2.1 for handling structured tensor operations.

Algorithm 1 (Sharing elimination)**.** The algorithm has three main phases: initialization (step 1); scheduling of multilinear operands for reaching quasi-optimality (steps 2-5); scheduling of all other sub-expressions (step 6).

(1) Perform a depth-first visit of the expression tree with recursive collection of identical sub-expressions.
Note: for example, an expression $(a+a+b+c+a)$ *would be transformed into* $(3a+b+c)$.
(2) Perform a depth-first visit of the expression tree to collect and partition multilinear operands into disjoint sets, $\mathbb{P} = \{P^1, ..., P^p\}$. $\mathbb{P}$ is such that multilinear operands in P share a set of multilinear symbols S_P, whereas there is no sharing across different partitions.
(3) For each $P \in \mathbb{P}$ such that $|P| \leq |S_P|$, apply strategy (i) to the multilinear operands.
Note: $|P|$ *and* $|S_P|$ *represent the number of products in the innermost loop induced by* P *if, respectively, strategies (i) or (ii) were applied. We will clarify that this directly descends from loop linearity.*

[3]In fact, it systematically appears in all of the forms used for experimentation (Section 6), and many more.

(4) Build the *sharing graph* $\mathbb{G} = (\mathbb{S}, \mathbb{E})$, with $S^i \in \mathbb{S}$ representing a multilinear symbol or a temporary produced at step 3. 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. Let $\mathbb{T}$ be the subset of vertices with a single incident edge, or "terminals".
(5) If $\mathbb{S} = \emptyset$, jump to step (6). Otherwise apply, in order, the following rewrite rules (*precondition*: *action*):
  (A) $\{T^1, ..., T^n\} \subset \mathbb{T}$ adjacent to S, $n > 1$: apply strategy (ii); $\mathbb{S} = \mathbb{S} \setminus \{T^1, ..., T^n\}$. *Note: strategy (ii) recasts the expression as* $S(...T^0 + ... + ...T^n)$.
  (B) Only one terminal T adjacent to S: apply strategy (ii) including one non-terminal symbol S^{nt}. $\mathbb{S} = \mathbb{S} \setminus \{T, S^{nt}\}$.
  (C) $\mathbb{T} = \emptyset$: take S adjacent to $\{S^1, ..., S^d\}$ such that d is maximum (i.e. S is the vertex with highest degree in $\mathbb{G}$) and apply strategy (ii); $\mathbb{S} = \mathbb{S} \setminus \{S^1, ..., S^d\}$.

  Go back to step (2).
(6) Perform a depth-first visit of the expression tree and, for each yet unhandled or hoisted sub-expression, apply the most profitable between strategies (i) and (ii). *Note: this pass speculatively assumes that structured tensor operations are in place. If the assumption does not hold, the result will generally be sub-optimal since only a subset of code motion opportunities may be exposed.*

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 (i) in step 3 has no effect. The sharing graph is $\mathbb{G} = (\{b_j, c_i, d_i\}, \{(b_j, c_i), (b_j, d_i)\}$, and $\mathbb{T} = \{c_i, d_i\}$. 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, the bilinear form arising from a Poisson equation in 2D. 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 (i) is applied to both partitions (step 3). We then have (step 4) $\mathbb{G} = (\{t^0, t^1, t^2, t^3\}, \emptyset)$, although no rewrite rule is applicable since $\mathbb{E} = \emptyset$.

```
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.

A much more complex example, using the hyperelastic model evaluated in Section 6, is made available online[4]

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

3.3. Monomial pre-evaluation

Sharing elimination explores the transformation space and applies three operators: expansion, factorization, and code motion. In this section, we discuss 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., specific factorization and code motion) preserving the correctness of the transformation.

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 can easily 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 opportunities can be exposed through exploration of the expression tree transformation space. This would be challenging if we were to deal with arbitrary loop nests and expressions. We make use of a result – the foundation of tensor contraction mode – to simplify our task. As summarized in Section 2.3, multilinear forms can be seen as sums of monomials, each monomial being an integral over the equation domain of products (of derivatives) of functions from discrete spaces; such monomials can always be reduced to a product of two tensors. This result can be turned into a transformation algorithm for loops and expressions, for which we provide a succinct description below.

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

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

(1) In contrast to what happens with hoisting in multilinear loop nests, the temporary variable size is proportional to the number and trip counts of non-reduction loops crossed (for the bilinear form implementation in Figure 1, JK for sub-expressions depending on $[L_i, L_j, L_k]$ and EJK for those depending on $[L_e, L_i, L_j, L_k]$). This might shift the loop nest from a CPU-bound to a memory-bound regime, which might be counter-productive for actual execution time.

(2) The transformations exposing $[L_i, L_j, L_k]$-dependent terms increase, in general, the arithmetic complexity (e.g., expansion may increase the operation count). This could outweigh the gain due to pre-evaluation.
(3) The need for a strategy to coordinate sharing elimination and pre-evaluation opportunities: sharing elimination inhibits pre-evaluation, whereas pre-evaluation generally exposes further sharing elimination opportunities.

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

3.4. Memory constraints

In the previous section, we provided an insight into the potentially negative effects of code motion. We now expand on this matter, starting with the following observations.

— The fact that $E \gg I, J, K$ suggests we should be cautious about hoisting mesh-dependent (i.e., L_e-dependent) expressions. Imagine Λ is enclosed in a time stepping loop. One could think of exposing (through some transformations) and hoisting any time-invariant sub-expressions to minimize redundant computation at every time step. The working set size could then increase by a factor E. The gain in number of operations executed could therefore be outweighed, from a runtime viewpoint, by a much larger memory pressure.
— For certain forms and discretizations, aggressive hoisting can make the working set exceed the size of some level of local memory (e.g. the last level of private cache on a conventional CPU, the shared memory on a GPU). For example, as explained in Section 3.3, pre-evaluating geometry-independent expressions outside of Λ requires temporary arrays of size JK for bilinear forms and of size J (or K) for linear forms. This can sometimes break such a "local memory threshold". In our experiments (Section 6.2) we will carefully study this aspect.

Based on these considerations, we establish two *memory constraints*.

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

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

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

3.5. Completeness under factorization, code motion, and reduction pre-evaluation

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

Sharing elimination and pre-evaluation tackle the problem of minimizing spatial and temporal redundancy given a set of memory constraints in finite element integration loop nests, using *a very specific set of operators*. This needs be emphasized since, theoretically, one could find an even lower operation count by exploiting domain-specific properties, such as redundancies in basis functions.

4. SELECTION AND COMPOSITION OF TRANSFORMATIONS

In this section, we build a transformation algorithm that, given a memory bound, produces optimal or 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) *Search for a global optimum.* 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 $\theta : M \to \mathbb{Z}$ 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 $\theta(m) = \theta_{se}(m) - \theta_{pre}(m)$, 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 focus on 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 subtle yet fundamental reasons for not pre-evaluating $m_1 \in P$: 1) the presence of spatial sharing between m_1 and $m_2 \in S$, which impacts the search for the global optimum; 2) breaking memory constraints.
(2) Build the set B of all possible bipartitions of P.
(3) For each $b = (b_S, b_P) \in B$, if the memory required to store the pre-evaluated tables from the monomials in b_P exceeds T_H (see Constraint 2), discard b; otherwise, add an entry to the dictionary d of potential operation counts, such that $d[b] = \theta_{se}(S \cup b_S) + \theta_{pre}(b_P)$.
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 b such that $min(d[b])$.
(5) Apply pre-evaluation to all monomials in $P \cup b_P$. Apply sharing elimination to the whole expression.
Note: because of reuse of basis functions, pre-evaluation may result in identical tables, which will be mapped to the same temporary. Sharing elimination is therefore transparently applied to the whole expression, including what results from pre-evaluation.

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

```
// 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.

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} is to apply the actual transformations and count the number of operations. If, on one hand, this is tolerable for θ_{se} (Algorithm 1 tends to have negligible cost), the overhead would be unacceptable if we applied pre-evaluation – in particular, symbolic execution – to all bipartitions analyzed by Algorithm 3. We then 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 enabling pre-evaluation. To contextualize, consider the example in Figure 6. One can think of this as the (simplified) loop nest originating from the integration of a pre-multiplied mass matrix. The sub-expression f_0*b_{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-dependent quantities. 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. Take the previous example. One can

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 have $n = 3$ (for b_{i0}, b_{i1}, and b_{i2}) and $k = 1$, which confirms what we intuitively found before, namely $\iota = 3$. 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 $\theta_{pre} = \rho \cdot \iota$. Consider again the mass matrix operator in Figure 6. The corresponding monomial is characterized by the dot product of test and trial functions, so trivially $\rho = 1$. If we instead take the example in Figure 3, we have that $\rho = 3$ after a suitable factorization of basis functions. In general, therefore, ρ depends on both form and discretization employed. To determine this parameter, we re-factorize the expression as per Algorithm 2 and simply count the terms amenable to pre-evaluation.

4.3. Formalization

The following proposition states that our approach guarantees quasi-optimality. The proof re-uses concepts and explanations provided throughout the paper, as well as the terminology introduced in Section 3.2.2.

Proposition 1. *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 pre-evaluated monomials, 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 9, and satisfies Constraints 1 and 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 correctly.

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 Constraints 1 and 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 skip a potential global optimum, since θ only reasons on 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 that share 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 local – instead of a global – optimum. This situation is prevented by construction, because Γ analyzes all possible bipartitions searching for a global optimum that also preserves the memory constraints. Recall that since the number of monomials is in practice very small, this pass can rapidly be accomplished. Finally, note how this problem can be seen as an instance of the well-known Knapsack problem.

(2) Consider Algorithm 1. Because of loop linearity and normal form of expressions, there are only two ways of scheduling the multilinear operands in $P \in \mathbb{P}$: through (i) generalized code motion or (ii) factorization of multilinear symbols. These two strategies lead, respectively, to performing $|P|$ or $|S_P|$ products at every loop iteration. Since (i) is side effects free (i.e., neither expansion nor factorization are needed) and is applied if and only if $|P| < |S_P|$, it is clear that step (3) cannot prune any optimal solution from the search space.

Steps (4)-(5) factorize symbols following the ordering dictated by the sharing graph and its rewrite rules. Recall that rules are applied in sequence: first all rule (A) hits are handled, then all (B)s and finally all (C)s, with applications of rule (C) that may expose further possibilities for (A) and (B). We prove that the application of these rules, in this sequence, represents a step towards quasi-optimality.

Rule (A). $\{T^0, ..., T^n\}$ can only be grouped through the factorization of S, so this rule is side effects free (i.e., no need to choose between different factorizations).

Rule (B). Since S^{nt} is no terminal, there must exist at least one edge between S^{nt} and a generic vertex V. Let us consider the alternative factorization $S^{nt}(...S + ... + ...V)$. If V is a terminal, we are back to where we started, because rule (B) will be applicable by setting $V = T$ and $S = S^{nt}$. Otherwise (V non terminal), we have S appearing in two different sub-expressions, which implies sub-optimality because of the presence of spatial sharing.

Rule (C). We can interpret G as a bipartite graph $\mathbb{B} = (\mathbb{S}_j, \mathbb{S}_k, \mathbb{E})$, in which $\mathbb{S}_j$ and $\mathbb{S}_k$ represent the partitions of symbols depending on L_j and L_k, respectively. For each disconnected component $\mathbb{B} = (\mathbb{S}_j^d, \mathbb{S}_k^d, \mathbb{E}^d)$ in $\mathbb{G}$, we consider two possibilities, namely $\mathbb{B}$ 1) is not or 2) is complete. Case 1) is the simplest: since, at this point, there are no more hits for rules (A) and (B), the lower bound on the operation count (as number of products performed at every loop iteration), lb, is given by $lb = min(|\mathbb{S}_j^d|, |\mathbb{S}_k^d|)$. This is reachable by factorizing the vertex with highest degree (as the rule states), as any other choice may break factorization opportunities. In case 2), we have $lb = 1$ because factorization is applicable iteratively (e.g., $a_j a_k + a_j b_k + b_j a_k + b_j b_k = (a_j + b_j)(a_k + b_k) = t_j t_k$); this is handled by going back to step (2), at the end of step (5).

The algorithm is guaranteed to terminate because of the jump to step (6) when $\mathbb{S} = \emptyset$, before applying the rewrite rules.

□

5. 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 discuss a few aspects concerning implementation and features of the compiler.

5.1. Automation 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 tiny, low level computer algebra system, not specifically 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. Sharing elimination and pre-evaluation are implemented by composing, in special ways, these operators.

5.2. Independence from form compilers

COFFEE is independent from the high level form compiler. COFFEE handles generic ASTs, so any form compiler can use it as long as such a representation is properly emitted (with expressions 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. COFFEE itself provides an interface for building an AST. In particular, COFFEE aims to decouple the mathematical manipulation of a form from code optimization. Another viewpoint is that a form compiler developers should not worry about performance of the generated code.

5.3. Handling block-sparse tables

Basis function tables may be block-sparse (e.g., containing zero-valued columns) depending on certain implementation choices at the level of the form compiler. For example, the FEniCS Form Compiler handles vector-valued function spaces by populating tabulated basis functions with blocks of zero-valued columns – this makes code generation easier, but performance sub-optimal due to the execution of "useless" flops (e.g., by executing operations like `a + 0`).

If, from one hand, it is true that form compilers usually provide optimization options to work around this issue, we must also consider the fact that not always their outcome is ideal for low level efficiency. Consider, for example, a set of tabulated basis functions with quadrature points along rows and functions along columns. `A[i,j]` provides the value of the `j-th` basis function at quadrature point `i`. In Ølgaard and Wells [2010], a technique to avoid iteration over zero-valued columns based on the use of indirection arrays (e.g. `A[B[i]]`, in which `A` is a tabulated basis function and `B` a map from loop iterations to non-zero columns in A) was proposed. This technique, however, promotes non-contiguous memory loads and stores, which nullify the benefits of vectorization.

COFFEE provides a mechanism to restructure loops that preserves the effectiveness of low level optimization, such as vectorization. By executing the code symbolically, loop bounds are adjusted and memory offsets are introduced to minimize "useless" flops. This is done performing several checks to preserve the semantics of the computation whilst trying not to affect low level optimization (e.g., padding and data alignment are affected by choice of the offsets – this is, however, out of the scope of this article).

As discussed in the next section, all code generation systems used for performance evaluation handle block-sparse tables.

6. PERFORMANCE EVALUATION

6.1. Experimental setup

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

We analyze the runtime performance of four real-world bilinear forms of increasing complexity, which comprise the differential operators that are most common in finite element methods. In particular, we study the mass matrix ("`Mass`") and the bilinear forms arising in a Helmholtz equation ("`Helmholtz`"), in an elastic model ("`Elasticity`"), and in a hyperelastic model ("`Hyperelasticity`"). The complete specification of these forms is made publicly available[5].

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

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

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

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

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

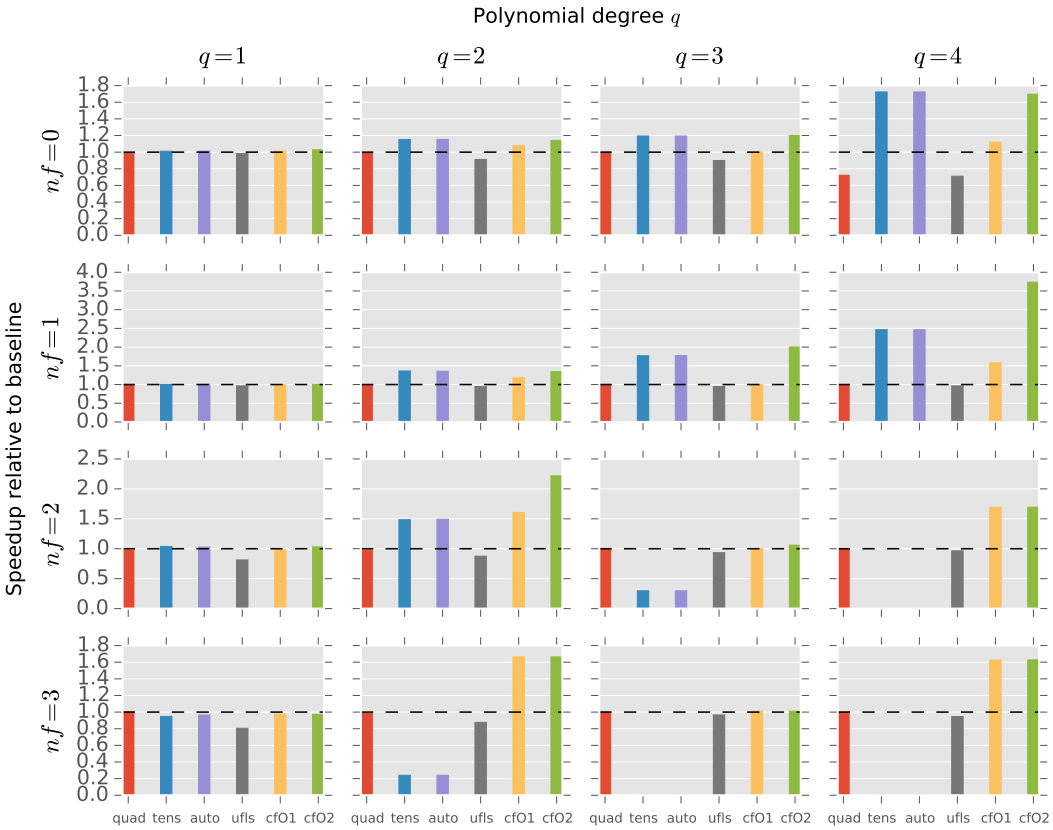

Fig. 7: Performance evaluation for the *mass* matrix. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

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

While constants of our study are

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

6.2. Performance results

We report the results of our experiments in Figures 7, 8, 9, and 10 as three-dimensional plots. The axes represent q, nf, and code transformation system. We show one subplot for each problem instance $\langle form, nf, q \rangle$, with the code transformation system varying within each subplot. The best variant for each problem instance is given by the tallest bar, which indicates the maximum speed-up over non-transformed code. We note that

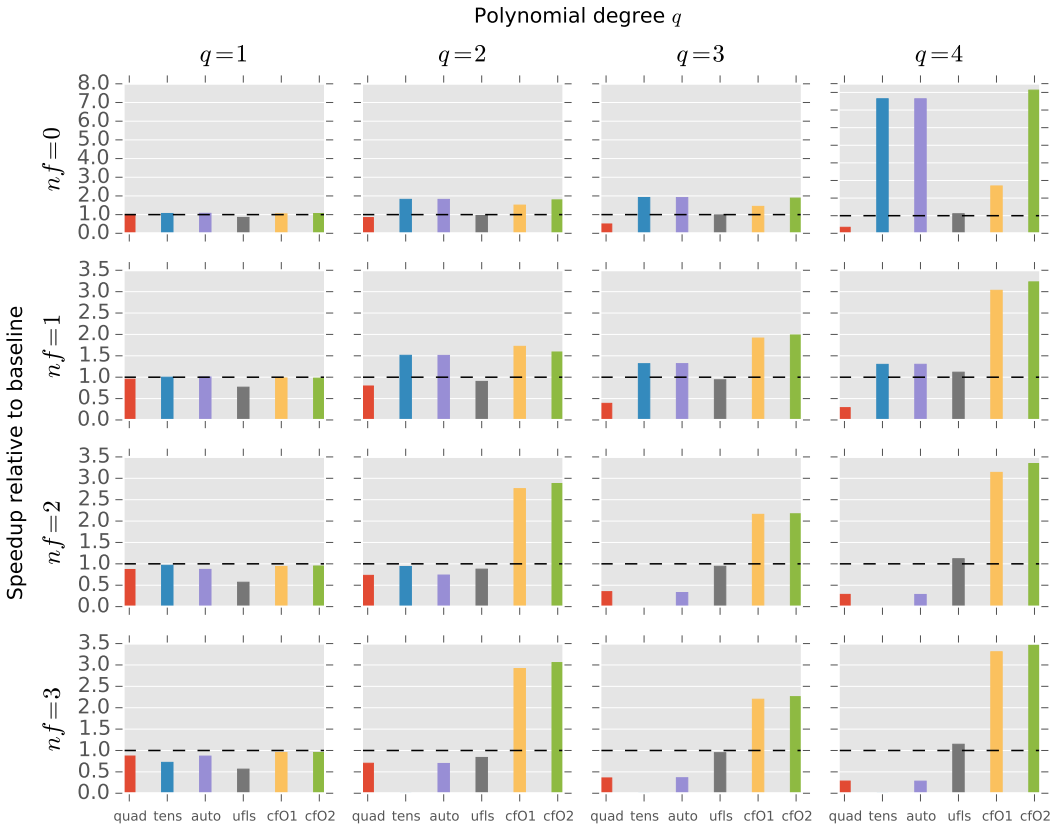

Fig. 8: Performance evaluation for the bilinear form of a *Helmholtz* equation. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

if a bar or a subplot are missing, then the form compiler failed at generating code because of either exceeding the system memory limit or unable to handle the form.

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

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

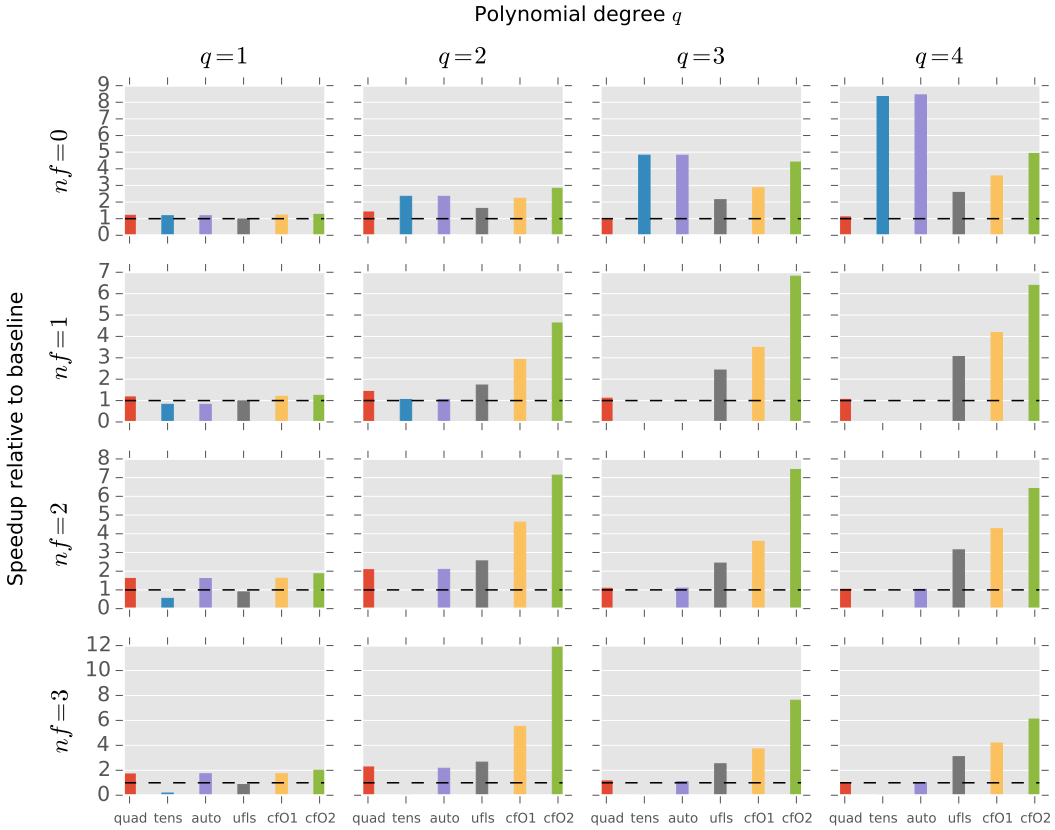

Fig. 9: Performance evaluation for the bilinear form arising in an *elastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

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

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

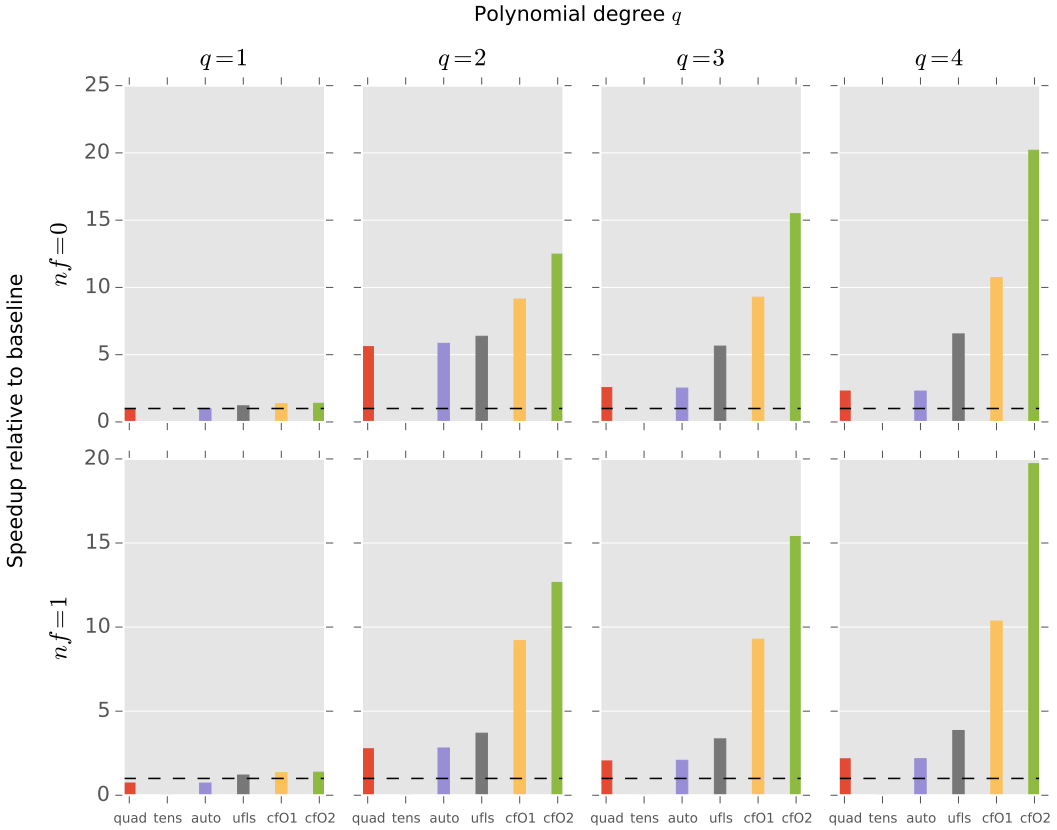

Fig. 10: Performance evaluation for the bilinear form arising in a *hyperelastic* model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

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.

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

7. CONCLUSIONS

With this research we have set the foundation of optimal finite element integration. We have developed theory and implemented an automated system capable of applying it. The automated system, COFFEE, is integrated in Firedrake, a real-world framework for writing finite element methods. We believe the results are extremely positive. An open problem is understanding how to optimally handle non-linear loop nests. A second open problem is extending our methodology to classes of loops arising in spec-

tral methods; here, the interaction with low level optimization will probably become stronger due to the typically larger working sets deriving from the use of high order function spaces. Lastly, we recall our work is publicly available and is already in use in the latest version of the Firedrake framework.

REFERENCES

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

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

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

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

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

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

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

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

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).