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 is defined in terms of floating point operations required to execute a loop nest. The generation of optimal loop nests is driven by a model that exploits mathematical properties of the domain of interest. A theoretical analysis and extensive experimentation prove the effectiveness of our approach, showing systematic performance improvements over a number of alternative code generation systems. The effect of low-level optimization is also discussed.

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), 21 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The need for rapidly implementing high performance, robust, and portable finite element methods has led to approaches based on automated code generation. This has been proved successful in the context of the FEniCS ([Logg et al. 2012]) and Firedrake ([Firedrake contributors 2014]) projects, which have become increasingly popular over the last years. 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 solution, are computed. The code for assembly operators should be high performance: as the complexity of a variational form increases, in terms of number of derivatives, pre-multiplying functions, or polynomial order of the chosen function

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

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

As demonstrated by the considerable body of research on the topic, automating the generation of such high performance implementations poses several challenges. This is a result of the complexity inherent to the mathematical expressions involved in the numerical integration, which varies from problem to problem, and the particular structure of the loop nests enclosing the integrals. General-purpose compilers, such as *GNU's* and *Intel's*, fail at exploiting the structure inherent in the expressions, thus producing sub-optimal code (i.e., code which performs more floating-point operations, or "flops", than necessary). Research compilers, for instance those based on polyhedral analysis of loop nests such as PLUTO ([Bondhugula et al. 2008]), focus on parallelization and loop optimization for cache locality, so they are not particularly helpful in our context. The lack of suitable third-party tools has led to the development of a number of domain-specific code transformation (or synthesizer) systems. In Olgaard 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 novel form compiler, called UFLACS ([Alnæs 2015]), which adds to the already abundant set of code transformation systems for assembly operators.

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

— We characterize flop-optimality in a loop nest and we instantiate this concept to finite element integration. As part of this construction, we establish the notion of sharing and demonstrate that sharing can always be eradicated from the loop nests we are interested in.
— We provide a model centred on sharing and other mathematical properties of our domain of interest; the model drives the translation of a monomial appearing in a form into flop-optimal loop nests.
— We comment on the cases in which such model does not lead to optimal loop nests. We will see that this might occur with particular forms that make extensive use of tensor algebra. We show, however, that our model still is capable of producing quasi-optimal loop nests.
— We integrate the model 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 of 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). We elaborate

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

on the performance impact of this optimization, making a clear distinction between flop optimality and efficient code.

2. PRELIMINARIES

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

We consider the weak formulation of a linear variational problem

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

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:

$$A\mathbf{u} = b \tag{2}$$

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 solution. In global assembly, such local contributions are "coupled" by suitably inserting them into A and b.

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

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

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

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

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

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

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

2.1. Quadrature Mode

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

The numerical integration routine based on quadrature over an element K can be expressed as follows

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

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.

2.2. Tensor Contraction Mode

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

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

A generalization of this transformation has been proposed in [Kirby and Logg 2007]. By 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

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

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

2.3. Qualitative Comparison

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

These two modes have been implemented in the FEniCS Form Compiler ([Kirby and Logg 2006]). In this compiler, a heuristic is used to choose the most suitable mode for a given form. It consists of analysing each monomial in the form, counting the number of derivatives and coefficient functions, and checking if this number is greater than a constant found empirically ([Logg et al. 2012]). We will later comment on the efficacy of this approach (Section 4. For the moment, we just recall that one of the goals of this research is to produce an intelligent 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. PROBLEM CHARACTERIZATION

In this section, we characterize optimality, as well as the transformation space that needs be explored to achieve it, for finite element integration.

3.1. Loop Nests and Optimality

In order to make the document 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 *linear loop* and, more generally, (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*

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

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

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

Definition 5 (Tensor-product loop nest)**.** *A tensor-product loop nest is a perfect multilinear loop nest $\Lambda = [L_{i_0}, L_{i_1}, ..., L_{i_{n-1}}]$ in which all of the expressions appearing in the body of $L_{i_{n-1}}$ are (summations of) tensor products $t_{i_0,i_1,...,i_{n-1}} = a_{i_0} b_{i_1} ... z_{i_{n-1}}$.*

As shown later, tensor-product loop nests are important because they possess properties that simplify the synthesis of optimal code. They arise naturally when translating a multilinear form into code. Consider Equation 7 and the loop nest implementing it illustrated in Figure 1. The imperfect nest $\Lambda = [L_e, L_i, L_j, L_k]$ comprises the element loop L_e, the integration loop L_i (a reduction loop), and the tensor-product loop-nest $[L_j, L_k]$ over test and trial functions. The expression `F` implements the operator.

```
for (e = 0; e < L; e++)
  ...
  for (i = 0; i < M; i++)
  ..
    for (j = 0; j < N; j++)
      for (k = 0; k < O; k++)
        a_ejk += F(...)
```

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

Our aim is to formulate a strategy to synthesize optimal implementations of such loop nests. In particular, we characterize optimality as follows.

Definition 6 (Optimality of a loop nest)**.** *The synthesis of a loop nest is optimal if the number of operations performed in the loop nest is minimal.*

Note that Definition 6 does not take into account memory requirements. If the loop nest were memory-bound – the ratio of operations to bytes transferred from memory to the CPU being too low – then speaking of optimality would clearly make no sense. Henceforth we assume to operate in a CPU-bound regime, in which arithmetic-intensive expressions need be evaluated. In the context of finite element, this is often true for more complex multilinear forms and/or higher order elements.

3.2. Transformation Space

To synthesize optimal implementations we need:

— a characterization of the transformation space for the class of loop nests considered
— a cost model to select the optimal point in the transformation space

In this section, we construct the transformation space. We leave a few loose hands, which we progressively tie in Section 4.

We start with introducing the fundamental notion of sharing.

Definition 7 (Sharing)**.** *A loop L_i presents sharing if it contains at least two expressions depending on i that are symbolically identical.*

Figure 2(a) shows an example of a trivial tensor-product loop nest of arity $n = 2$ with sharing along dimension j.

```
for (j = 0; j < O; j++)
  for (i = 0; i < N; i++)
    a_ji += b_j c_i + b_j d_i
```

(a) With sharing

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

(b) Optimal form

Fig. 2: A simple tensor-product loop nest

We prove that tensor-product loop nests can be reduced to optimal form (i.e., the number of operations in the resulting loop nest is minimal) by eliminating sharing through a suitable algorithm.

Proposition 1. *Assume $\Lambda = [L_{i_0}, L_{i_1}, ..., L_{i_{n-1}}]$ is a tensor-product loop nest with the body of $L_{i_{n-1}}$ being an independent basic block. Then we can determine an optimal Λ' applying a sequence of steps aiming at eliminating sharing.*

Proof. The demonstration is by construction and exploits linearity (recall that tensor-product loop nests are multilinear by definition). We start with "flattening" the expressions appearing in the body of $L_{i_{n-1}}$; that is, products are recursively expanded. This exposes the space of all possible factorizations. The terms appearing in the expanded expressions are tensor products of the form $a_{i_0} b_{i_1} ... z_{i_{n-1}}$ (a tensor product involves n arrays , each array indexed by a linear loop, or simply symbols).

We logically group the symbols into n disjoint sets S_i, each S_i containing the symbols indexed through L_i. These sets are sorted in ascending order based on their cardinality. Based on this, we create the factorization list: the ordered list of all factorizable

symbols. We then progressively try to factorize these symbols. That is, we start factorizing along the loop that has the least number of unique symbols, and then we go on with the other loops.

Due to linearity, each factored product only has one symbol depending on L_i, and this symbol is now unique in the expression. The other symbols in the term, independent of L_i, are, by definition, loop-invariant, and therefore hoisted such that redundant computation is avoided. This procedure is always semantically correct, regardless of the symbols (loops) being considered: multilinearity ensures deterministic factorization; perfectness ensures safeness of hoisting.

Two observations related to the aforementioned code motion: 1) hoisting might require the creation of "clone loops" outside of Λ to honor the potential dependency on loop indices; 2) hoisted expressions are replaced with temporaries, which need be added to a suitable set S_i of factorizable symbols .

It remains to clarify why the chosen factorization ordering leads to optimality. Naturally, as we factorize a symbol, we are potentially pruning other factorization opportunities. If that is the case, we might wonder whether there was a better factorization strategy. . Therefore, this is not possible (in the best case, the resulting factorization would be identical).

Same cardinality, try both □

For example, by applying this procedure to the code in Figure 2(a), we obtain the optimal form in Figure 2(b).

However, we observe that, for the larger class of finite element integration loop nests, the presence of sharing is a *sufficient but not necessary* condition for being in *non-optimal* form. Consider again the bilinear form implementation in Figure 1. We pose the following question: are we able to identify sub-expressions within F for which the reduction imposed by L_i can be pre-evaluated, thus obtaining a decrease in operation count proportional to the size of L_i, M? The transformation we look for is exemplified in Figures 3(a) (input) and 3(b) (output); Figure 3(a) can be seen as a simple instance of the abstract loop nest in Figure 1.

```
for (e = 0; e < L; e++)
  for (i = 0; i < M; i++)
    for (k = 0; k < O; k++)
      a_ek += d_e b_ik c_i + d_e b_ik d_i
```

(a) With reduction

```
for (i = 0; i < M; i++)
  for (k = 0; k < O; k++)
    t_k += b_ik (c_i + d_i)

for (e = 0; e < L; e++)
  for (k = 0; k < O; k++)
    a_ek = d_e t_k
```

(b) Pre-evaluated reduction

Fig. 3: Exposition (through factorization) and pre-evaluation of a reduction.

Pre-evaluation opportunities could have to be exposed, for instance through an exploration of the expression tree transformation space, which can be challenging. Further, the following issues arise when considering pre-evaluation opportunities:

— as opposed to what happens with hoisting in tensor-product loop nests, the temporary variable size is proportional to the number of non-reduction loops crossed (for the bilinear form implementation in Figure 1, $N \cdot O$ for sub-expressions depending on $[L_i, L_j, L_k]$ and $L \cdot N \cdot O$ 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 runtime

— the transformations exposing pre-evaluation opportunities could increase the arithmetic complexity (e.g., expansion may increase the operation count; further examples will be provided later). This could overwhelm the gain inherent in pre-evaluation.

To summarize, so far we have highlighted four problems:

(1) The need for an algorithm to expose pre-evaluation opportunities
(2) Potential explosion in working set size
(3) Potential increase in operation count due to manipulation of the expression tree
(4) The need for a strategy to coordinate sharing elimination and pre-evaluation opportunities

We tackle these points in Section 4. In the perspective of addressing point 2, it is useful to conclude this section refining our optimality statement as follows.

Definition 8 (Optimality of a loop nest with bounded working set)**.** *The synthesis of a loop nest is optimal if, under a set of memory constraints C, the number of operations performed in the loop nest is minimal.*

4. OPTIMAL INTEGRATION ROUTINES

In this section, we will reason at two different levels of abstraction: the math, in terms of the multilinear forms arising from the weak variational formulation of a problem; and the (partly multilinear) loop nests implementing such forms.

Our point of departure is the example loop nest in Figure 1. We make the following observations. 1) $L \gg M, N, O$; that is, the number of elements L is typically order of magnitude larger than both quadrature points (M) and degrees of freedom (N and O for test and trial functions); 2) $[L_j, L_k]$ (or simply L_j with a linear form) is perfect and multilinear; this naturally descends from the translation of Equation 7 into a loop nest.

4.1. Memory constraints

The fact that $L \gg M, N, O$ suggests we should be cautious about hoisting out of Λ. Imagine Λ is enclosed in a time stepping loop. One could think of exposing and then hoisting time-invariant sub-expressions. For those sub-expressions that depend on the mesh geometry (i.e., they would depend on L_e), code motion would increase the working set by a factor L. The gain in number of operations executed could then be overwhelmed, from a runtime viewpoint, by a much larger memory pressure.

A second, more general fact is that, for certain forms and discretizations, excessive 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, applying tensor contraction mode, which essentially means pre-evaluating geometry-independent expressions outside of Λ, requires temporary arrays of size $N \cdot O$. This can sometimes break the local memory threshold. We will clarify and experiment this in Section 6.2.

Based on these considerations, we characterize the set of memory constraints C (see Definition 8) as follows

(1) The size of a temporary due to code motion cannot be larger than that of the multilinear iteration space.
(2) The total amount of memory occupied by the hoisted temporaries cannot exceed a threshold T_H

A corollary of C_1 is that hoisting expressions involving geometry-dependent terms outside of Λ, which we discussed at the beginning of this section, is now forbidden. Consequently, the search space for optimality becomes smaller.

4.2. Minimizing the Operation Count in Inner Loops

Definition 8 states that a necessary condition for a loop nest synthesis to be optimal is that the sum of the innermost loop operation counts is minimum. We now discuss how we can systematically achieve this.

Eliminating sharing from the inner multilinear loops does not suffice. In fact, as suggested in Section 3, we wonder whether, and under what transformations, the reduction imposed by L_i could be pre-evaluated, thus reducing the operation count.

To answer this question, we make use of a result – the foundation of tensor contraction mode – from Kirby and Logg [2007]. Essentially, 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 (see Section 2). We interpret this result at the loop nest level: with an input as in Figure 1, we can always dissect `F` into distinct sub-expressions (the monomials). Each sub-expression is then factorized so as to split constant from $[L_i, L_j, L_k]$-dependent terms, the latter ones are hoisted outside of Λ, and finally pre-evaluated into temporaries. As part of this pre-evaluation, the reduction induced by L_i vanishes. In the following, we simply refer to this special sort of code hoisting as "*pre-evaluation*".

The challenge is to understand when, and for which monomials, pre-evaluation is profitable. We propose an algorithm and a discussion of its optimality. The intuition of the main algorithm for reducing a loop nest to optimal form is shown in Figure 4.

```
1  dissect the input expression into monomials
2  for each monomial M:
3    θ_w = estimate operation count with pre-evaluation
4    θ_wo = estimate operation count without pre-evaluation
5    if θ_w < θ_wo and memory constraints satisfied:
6      mark M as candidate for pre-evaluation
7  for each monomial M:
8    if M does not share terms with M', an unmarked monomial:
9      extract M into a separate loop nest
10     apply pre-evaluation to M
11 for each expression:
12   remove sharing
```

Fig. 4: Intuition of the main algorithm

We study the impact of pre-evaluation, as number of operations saved or introduced, "locally"; that is, for each monomial, in isolation. If we estimate that, for a given monomial, pre-evaluation will decrease the operation count, then the corresponding sub-expression is extracted, a sequence of transformation steps – involving expansion, factorization, code motion – takes place (details in Section 5), and the evaluation eventually performed. The result is a set of n-dimensional tables (these can be seen as "slices" of the reference tensor at the math level), n being the arity of the multilinear form. Identical tables are mapped to the same temporary. Eventually, sharing is removed from the resulting expressions by applying a procedure as described in Proposition 1. The transformed loop nest is as in Figure 5.

```
// Pre-evaluated tables
...
for (e = 0; e < L; e++)
  // Loop nest for pre-evaluated monomials
  for (j = 0; j < N; j++)
    for (k = 0; k < O; k++)
      A[e,j,k] += G'(...) + G''(...) + ...

  // Loop nest for monomials for which run-time
  // integration is preferable
  for (i = 0; i < M; i++)
  ..
    for (j = 0; j < N; j++)
      for (k = 0; k < O; k++)
        A[e,j,k] += H(...)
```

Fig. 5: A possible loop nest for the local assembly of a form after applying pre-evaluation to some of the monomials (resulting in sub-expressions `G'`, `G''`, ...)

Before elaborating on the profitability of pre-evaluation (i.e., how to determine the cost function θ in Figure 4), we need to discuss under which conditions this approach, based on a "local analysis" of monomials, is able to reduce loop nests to optimal form.

Proposition 2. *Consider an integration routine originating from an arbitrary multilinear form. A loop nest Λ, as previously characterized, evaluates an expression comprising a set of monomials, M. Let P be the set of monomials for which pre-evaluation is applied because found profitable, and let Z be the set of non pre-evaluated monomials (i.e., $Z = M \setminus P$). Assume that:*

(*1*) *the cost function θ employed is optimal; that is, it predicts correctly whether pre-evaluation is profitable or not for a monomial*
(*2*) *pre-evaluating different monomials does not result in identical tables*
(*3*) *monomials in P do not share terms*

Then, the main algorithm in Figure 4 leads to inner optimality under the set of memory constraints C (defined in Section 4.1).

Proof. We first comment on the assumptions. 1) We postpone the discussion of how to build θ to Section 4.3. 2) Identical pre-evaluated tables from distinct monomials could be the result of symmetries in tabulated basis functions. This is however a pathological case that we can harmlessly ignore. 3) In complex forms with several monomials, different pre-evaluation candidates could actually share terms. We abstract from this case, which otherwise would require a "global" analysis of the monomials in the form that we believe would not justify the gain (if any) in operation count.

We distinguish two classes of loop nests rooted in Λ: $[L_e, L_j, L_k]$, for the pre-evaluated monomials in P, and $[L_e, L_i, L_j, L_k]$, enclosing the remaining monomials in Z. Since they only differ for the presence of L_i, we relieve the notation by omitting the shared loops when discussing operation counts. The operation count of what we are proving to be the optimal Λ synthesis (inferable from Figure 5) is $\Lambda_{ops} = \Lambda_{ops_1} + \Lambda_{ops_2} = \sum_{\alpha}^{\#P} p_\alpha + I \sum_{\beta}^{\#Z} z_\beta$, where p_α and z_β represent the operation count of monomials in P and Z, respectively, while I is the iteration space size of L_i.

We note that, as explained in Section 4.1, C imposes constraints on hoisting. This narrows the proof to demonstrating the following: A) pre-evaluating any $Z_P : Z_P \subseteq Z$ would increase Λ_{ops}; B) not pre-evaluating any $P_Z : P_Z \subseteq P$ would increase Λ_{ops}.

A) We prove that $\Lambda'_{ops} = \Lambda'_{ops_1} + \Lambda'_{ops_2} > \Lambda_{ops}$. It is rather obvious that $\Lambda'_{ops_1} \geq \Lambda_{ops_1}$ (it is equal only if, trivially, $Z_P = \emptyset$). We note that if monomials in Z_P share terms with $\overline{Z} = Z \setminus Z_P$, then we have $\Lambda'_{ops_2} = \Lambda_{ops_2}$, so our statement is true. If, on the other hand, at least one monomial does not share any terms, we obtain $\Lambda'_{ops_2} < \Lambda_{ops_2}$ or, equivalently, $\Lambda'_{ops_2} = \Lambda_{ops_2} - I \cdot \delta$. What we have to show now is that even by exposing more pre-evaluations, $\Lambda'_{ops_1} - \Lambda_{ops_1} = \gamma \geq I \cdot \delta$ holds. Because of assumption 2), the new pre-evaluations cannot expose further sharing. Therefore, the optimality of the cost function (assumption 1) ensures our claim holds.

B) In absence of sharing, the statement is trivially true since we would have $\Lambda'_{ops_2} > \Lambda_{ops_2}$, being the cost function optimal due to assumption 1). Assumption 3) guarantees there cannot be sharing within P_Z, which avoids subtle cases in which pre-evaluation would be sub-optimal due to destroying sharing-removal opportunities. The last case we have to consider is when $p \in P_Z$ shares at least one term with $z \in Z$. This situation cannot actually occur by construction: all candidates for pre-evaluation sharing terms with monomials in Z are "de-classed" from P to Z (see Figure 4, line 8). The rationale is that since we would have to pay anyway the presence of z in the innermost loop, adding p to Z would not augment the operation count in our model, so we can safely avoid pre-evaluation.

□

4.3. A-Priori Operation Counting

It remains to tie one loose hand: the construction of the pre-evaluation cost function θ. We define it such that $\theta : M \rightarrow \mathbb{N} \times \mathbb{N}$; that is, given a monomial, two natural numbers representing the sharing-free operation count without (θ_{wo}) and with (θ_w) pre-evaluation are returned. Since θ is expected to be used by a compiler to drive the transformation process, requirements are simplicity and velocity.

We can easily predict θ_{wo} thanks to our key property, linearity. This was explained in Proposition 1. A simple analysis suffices to obtain the cost of a sharing-free multilinear loop nest, namely Λ^{sf}_{ops}. Assuming I to be the size of the L_i iteration space, we have that $\theta_{wo} = \Lambda^{sf}_{ops} \cdot I$.

For θ_w, things are more complicated. We first need to estimate the *increase factor*, ι, to account for the presence of (derivatives of) coefficients. This number captures the increase in arithmetic complexity due to the transformations enabling pre-evaluation. To contextualize, consider the example in Figure 6.

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

Fig. 6: Simplified loop nest for the local assembly of a pre-multiplied mass matrix.

One can think of this as the (simplified) loop nest originating from the assembly of a pre-multiplied mass matrix. The sub-expression `f[0]*B[i,0]+f[1]*B[i,1]+f[2]*B[i,2]` represents the field f over (tabulated) basis functions B. In order to apply pre-evaluation, the expression needs be transformed to separate f from the integration-dependent (i.e., L_i-dependent) quantities. By expanding the product we observe an increase in the number of $[L_j, L_k]$-dependent operations of a factor 3 (the local degrees of freedom for the coefficient). Intuitively, ι captures this growth in non-hoistable operation.

With a single coefficient, as we just saw, ι directly descends from the cost of expansion. In general, however, predicting ι is less straightforward. For example, consider the case in which a monomial has multiple coefficients expressed over the same function space. The expansion would now lead to identical sub-expressions that, once pre-evaluated, would be mapped to the same temporary. The resulting loop nest would be characterized by sharing, as a result. Therefore, the actual operation count (i.e., once sharing is removed) would be smaller than that one could infer from analysing the expansion "in isolation". For a precise estimate of ι, we then need to calculate the k-combinations with repetitions of n elements, with k being the number of coefficient-dependent terms appearing in a product (in the example, there is only f, so $k = 1$) and n the cardinality of the set of symbols involved in the coefficient expansion (in the example, `B[i,0]`, `B[i,1]`, and `B[i,2]`, so $n = 3$; note that we are talking about sets here, so duplicates would be counted once).

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

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

5. CODE GENERATION

The model described in Section 4 has been fully automated in COFFEE, the optimizer of finite element integration routines used in Firedrake. In this section, we describe the features of this code generation system.

5.1. Automation through the COFFEE Language

As opposed to what happens in the FEniCS Form Compiler with quadrature and tensor modes, there are no separate trunks in COFFEE handling pre-evaluation, sharing, and code motion in general. All optimizations are instead expressed as composition of parametric "building-block" operations. 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: other domains sharing properties similar to that of finite element integration (e.g., multilinear loop nests) could be optimized through the same compiler. Thirdly, robustness: the same building-block operations are exploited, and therefore stressed, by different optimization pipelines.

A non-exhaustive list of such operations includes expansion, factorization, re-association, generalized code motion. These "rewrite operators" can be seen as the COFFEE language. They define parametric transformations: for example, one could ask to factorize constant rather linear terms, while hoisting could be driven by loop dependency. Their implementation is based on manipulation of the abstract syntax tree representing the integration routine.

5.1.1. Heuristic Optimization of Integration-dependent Expressions. As a proof-of-concept of our generality claim, we briefly discuss our optimization strategy for integration-dependent expressions. These are expressions that should logically be placed within L_i. They can originate, for example, from the extensive use of tensor algebra in the derivation of the weak variational form or from the use of a non-affine reference-to-physical element mapping, which Jacobian needs be re-evaluated at every quadrature point. For some complex monomials and for coarser discretizations, the operation count within L_i could be comparable or, in some circumstances, even outweigh that of the multilinear loop nest. In these cases, our optimality model becomes weaker, since its underlying assumption is that the bulk of the computation is carried out in innermost loops.

Despite the fact that we are not characterizing optimality for this wider class of problems, we can still heuristically apply the same reasoning of Sections 3 and 4 to try to eliminate sharing, thus reducing the operation count. This is straightforward in our code generation system by composing rewrite operators. Our strategy is as follows.

COFFEE is agnostic with respect to the high level form compiler, so the first step consists of removing redundant sub-expressions. This is because a form compiler abstracting from optimization will translate expressions as in Equation 7 directly to code without performing any sort of analysis. Eliminating redundant sub-expressions is usually helpful to relieve the arithmetic pressure inherent in L_i. We then synthesize an optimal loop nest as described in the previous sections. This may in turn expose a set of L_i-dependent expressions. For each of this expressions, we try to remove sharing by greedily applying factorization and code motion. In the COFFEE language, this process is expressed by simply composing five rewrite operators.

5.2. Low-level Optimization

We comment on a set of low-level optimizations. These are essential to 1) achieve machine peak performance (Sections 5.2.1 and 5.2.2) and 2) make COFFEE independent of the high-level form compiler (Section 5.2.3). As we will see, there are interplays among different transformations. For completeness, we present all of the transformations available in the compiler, although we will only use a subset of them for a fair performance evaluation.

5.2.1. Review of Existing Optimizations. We start with briefly reviewing the low level optimizations presented in Luporini et al. [2015].

Padding and data alignment. All of the arrays involved in the evaluation of the local element matrix or vector are padded to a multiple of the vector register length. This is a simple yet powerful transformation that maximizes the effectiveness of vectorization. Padding, and then loop bounds rounding, enable data alignment and avoid the introduction of scalar remainder loops.

Vector-register Tiling. Blocking (or tiling) at the level of vector registers improves data locality beyond traditional unroll-and-jam transformations. This blocking strategy consists of evaluating outer products by using just two vector register and without ever spilling to cache.

Expression Splitting. When the number of basis functions arrays (or, equivalently, temporaries introduced by code motion) and constants is large, the chances of spilling to cache are high in architectures with a few logical registers (e.g. 16/32). By exploiting sums associativity, an expression can be fissioned so that the resulting sub-expressions can be computed in separate loop nests. This reduces the register pressure.

5.2.2. Vector-promotion of Integration-dependent Expressions. Integration-dependent expressions are inherently executed as scalar code because vectorization (unless employing special hand-written schemes) occurs along a single loop, typically the innermost. For the same reasons discussed in Section 5.1.1, we also want to vectorize along L_i. One way to achieve this is vector-promotion. This requires creating a "clone" of L_i in the preheader of the loop nest, in which vector temporaries are evaluated in what is now an innermost loop.

5.2.3. Handling Sparse Tables. Consider a set of tabulated basis functions with quadrature points along rows and functions along columns. For example, `A[i,j]` provides the value of the `j-th` basis function at quadrature point `i`. Unless using a smart form compiler (which we want to avoid), there are circumstances in which the tables are sparse. Zero-valued columns arise when taking derivatives on a reference element or when employing vector-valued elements. Zero-valued rows can result from using non-standard functions spaces, like Raviart-Thomas. Zero-valued blocks can appear in pre-evaluated temporaries. Our objective is a transformation that avoids useless iteration over zeros while preserving the effectiveness of the other low-level optimizations, especially vectorization.

In Olgaard and Wells [2010], a technique to avoid iteration over zero-valued columns based on the use of indirection arrays (e.g. `A[B[i]]`, in which `A` is a tabulated basis function and `B` a map from loop iterations to non-zero columns in A) was proposed. Our approach, which will be compared to this pioneering work, aims to free the generated code from such indirection arrays. This is because we want to avoid non-contiguous memory loads and stores, which can nullify the benefits of vectorization.

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

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

6. PERFORMANCE EVALUATION

6.1. Experimental Setup

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

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

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

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

— FEniCS Form Compiler: optimized quadrature mode (work presented in Olgaard and Wells [2010]). Referred to as `quad`
— FEniCS Form Compiler: tensor mode (work presented in Kirby and Logg [2006]). Referred to as `tens`
— FEniCS Form Compiler: automatic mode (choice between `tens` and `quad` driven by heuristic, detailed in Logg et al. [2012] and summarized in Section 2.3). Referred to as `auto`
— UFLACS: a novel back-end for the FEniCS Form Compiler (whose primary goals are improved code generation time and runtime). Referred to as `ufls`
— COFFEE: generalized loop-invariant code motion and padding (work presented in Luporini et al. [2015]). Referred to as `cf01`
— COFFEE: optimal multilinear loop nest synthesis, padding and symbolic execution (captures the contributions of this paper). Referred to as `cf02`

The values that we report are the average of three runs with "warm cache" (no code generation time, no compilation time). They include the cost of local assembly as well as the cost of matrix insertion. However, the unstructured mesh used to run the simulations 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 the fairest comparison possible, small patches (publicly available) were written to be able to run all simulations through Firedrake: this means the cost of matrix insertion and mesh iteration is exactly the same in all variants. UFLACS and the FEniCS Form Compiler's optimization systems generate code suitable for FEniCS, which employs a data storage layout different than Firedrake. Our patches fix this problem by producing code with a data storage layout as expected by Firedrake.

For what concerns the memory constraint C_2 introduced in Section 4.1, we set $T_H = L2_{size}$; that is, the amount of space that temporaries due to pre-evaluation can occupy is bounded by the size of the processor L2 cache (the last level of private cache). For the test cases in which T_H was determinant to prevent pre-evaluation, we also repeated the experiments setting $T_H = L3_{size}$ to assess our hypotheses. Results of these trials are discussed below.

Following the methodology adopted in Olgaard and Wells [2010], we increasingly vary the complexity of each form. In particular:

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

Constants of our analysis, instead, are

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

The upcoming Figures (7, 8, 9, and 10) can be interpreted as "nested" plots. We refer to the outer plot as the "grid", while the inner are the actual "plots". In a grid, p varies along the horizontal axis and q varies along the vertical axis. The top-left plot in a grid shows the speed-up over original code for $[q = 1, p = 1]$; the plot on its right for $[q = 1, p = 2]$, and so on. The diagonal of the grid provides the behaviour when test,

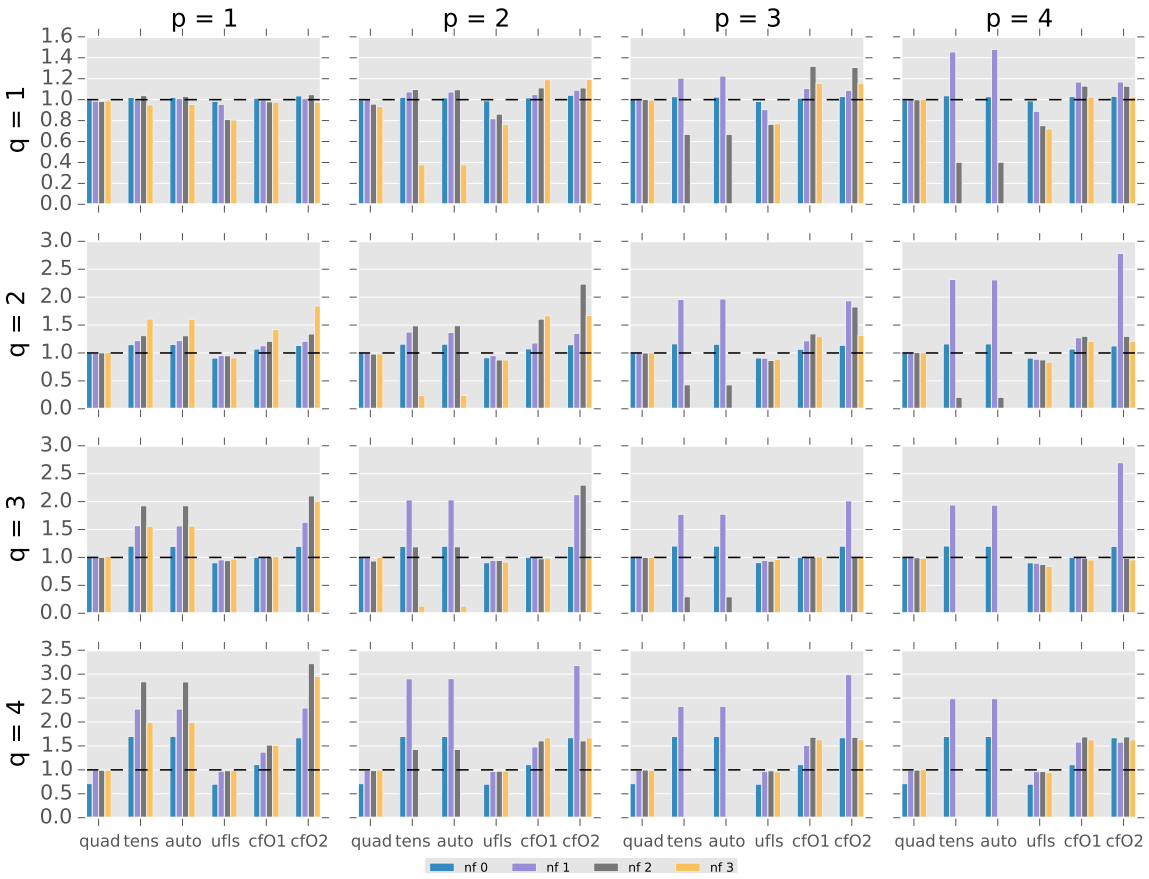

Fig. 7: Performance evaluation for the mass matrix. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

trial and coefficient functions have same polynomial order, that is $q = p$. A grid can therefore be looked at from different perspectives, which allows us to make structured considerations on the performance achieved. In each plot there are six groups of bars, each group referring to a particular code variant (`quad`, `tens`, `...`). There are four bars per group: the leftmost bar corresponds to the case $nf = 0$, the one on its right to the case $nf = 1$, and so on.

6.2. Performance of Forms

The first observation is that our optimality model does not imply minimum execution time. In particular, in the 0.06% of the test cases (we are not counting marginal differences), `cfO2` does not result in the best runtime. There are two reasons for this. The former is that low level optimization can have a significant impact. Compare, as an example, `tens` and `cfO2` in `Mass` and `Helmholtz` when $q = 1$. With $p \in [2, 3]$, `tens` slightly outperforms `cfO2`. With $p = 4$, `tens` is quite faster than `cfO2`. The motivation is the aggressive unrolling performed in `tens`, which 1) allows to eliminate all operations involving zero-valued entries and 2) can drastically decrease both working set and register pressure by avoiding storing entire temporaries (there are often many re-

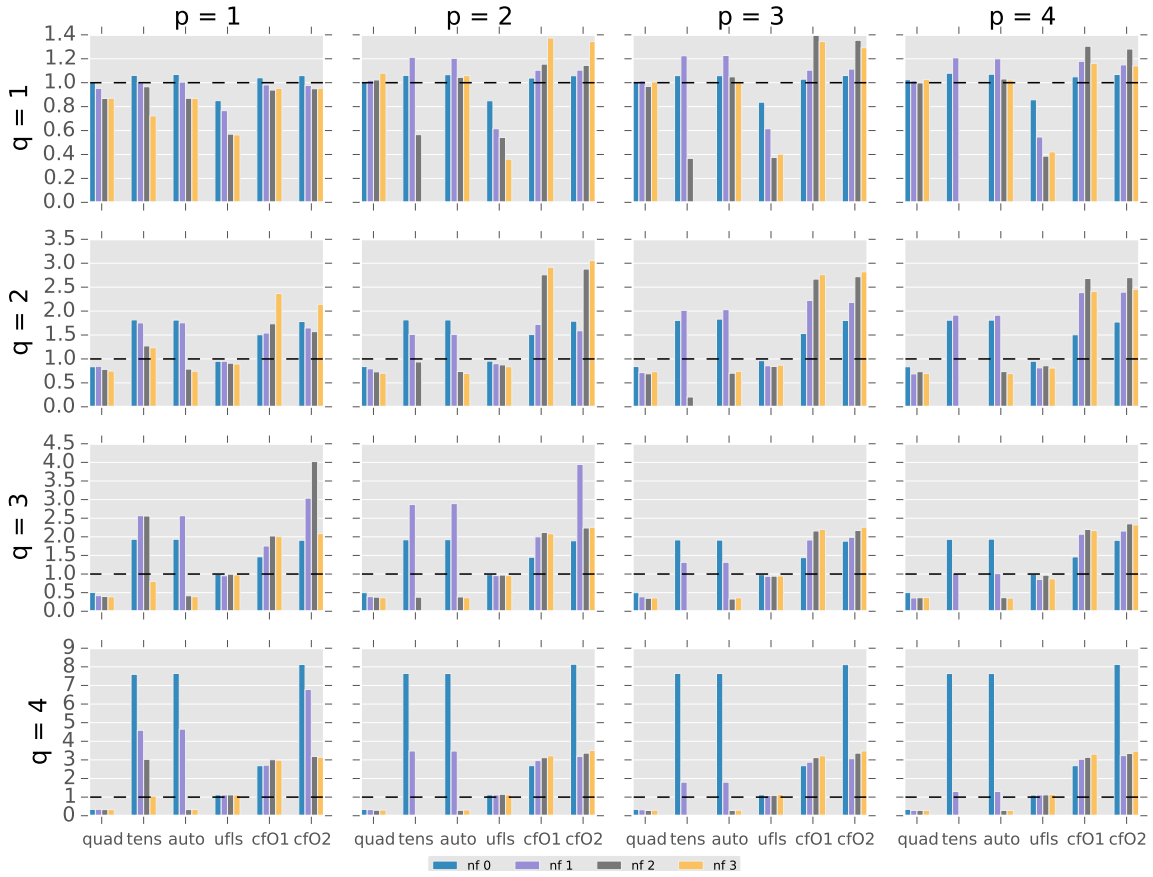

Fig. 8: Performance evaluation for the bilinear form of a Helmholtz equation. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

peated values that, with unrolling, need be allocated only once). In COFFEE we never unroll loops to maximize the chances of autovectorization, which usually results in considerable speed-ups throughout the majority of test cases. The latter reason is simply inherent to our optimality model. As explained in Section 5.1.1, we use heuristics to optimize the integration loop. The reason we are studying an hyperelastic model is indeed to assess their effectiveness; the systematic performance improvements observed over all other variants are, however, extremely encouraging.

The second aspect regards the "triggering" of pre-evaluation in `cf02`. The fact that a bar in `cf02` matches the corresponding bar in `cf01` usually suggests that pre-evaluation was found unprofitable. The trend – although there are quite a few exceptions – is that as nf increases, pre-evaluation tends not to be performed. This is a consequence of a larger increase factor, as discussed in Section 4.3.

The third point concerns the effect of the chosen discretizations on autovectorization. The discretizations employed ensure both multilinear loop nest and tabulated basis function sizes to be multiple of the machine vector length. Given the suitable choice of compilation options, this promotes autovectorization in the majority of code variants.

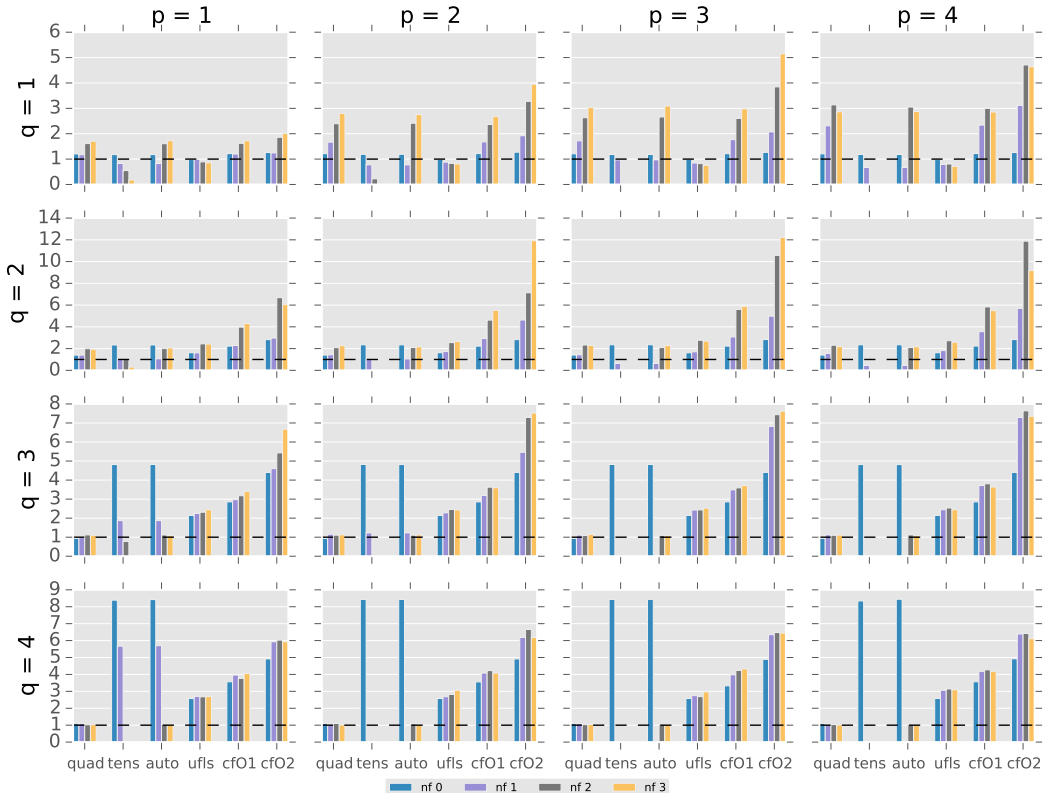

Fig. 9: Performance evaluation for the bilinear form arising in a elastic model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

The biggest exception is quad, in which, due to the presence of indirection arrays in the generated code, vectorization is rarely triggered. In `tens`, the loop nests are fully unrolled, so the standard loop vectorization is simply not possible; however, manual inspection of the compiled code suggests block vectorization ([Larsen and Amarasinghe 2000]) is often triggered. In `ufls`, `cf01`, and `cf02` the iteration spaces tend to have the same "structure" (there are very few exceptions due to low level optimization), with loop vectorization being regularly applied, as far as we could evince. It is worth noting the fact that the size of loops and tabulated basis functions is a multiple of the vector length render padding and data alignment (in `cf01` and `cf02`), to the best of our understanding, basically irrelevant.

Also note some other minor things.

— The lack of (or marginal) improvements in all code variants when $[q = 1, p = 1]$ is due to the cost of matrix insertion, which outweighs that of local assembly.
— The slow-downs experienced with quad are imputable to the presence of indirection arrays in the generated code; especially with dense tables, trading vectorization for avoiding iteration over a few zero-valued columns may be counter-productive (see also Section 5.2.3).
— A missing bar in `tens` or `auto` means that the code generation system failed because of either exceeding the memory limit or being unable to manipulate the math char-

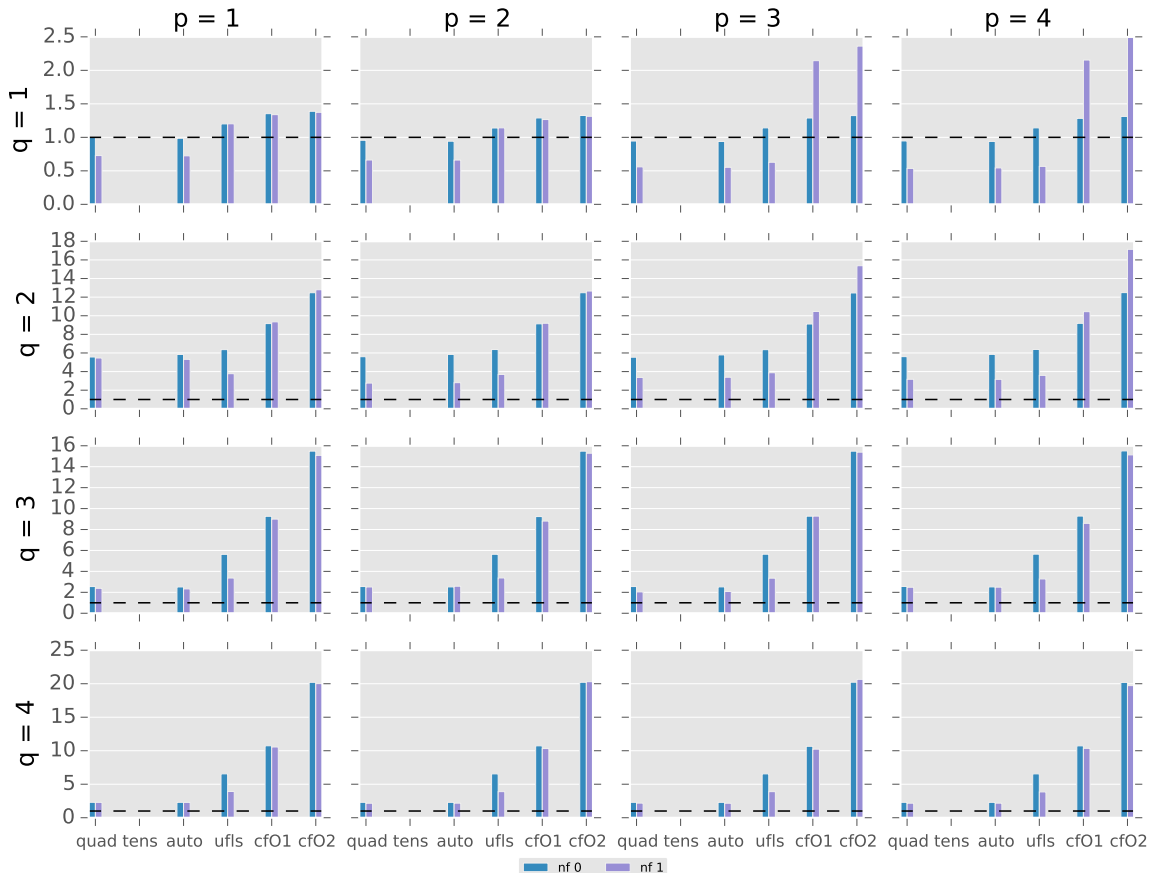

Fig. 10: Performance evaluation for the bilinear form arising in a hyperelastic model. The bars represent speed-up over the original (unoptimized) code produced by the FEniCS Form Compiler.

acterizing the form (this happens systematically in the hyperelastic model, and for some discretizations in the elastic model).

— The bars for $nf = 0$ do not change as p increases: this is expected since varying p when there are no coefficients should not make any difference.

Mass. From analysis of Figure 7, we can see that `cf02` is the best variant among the ones tested, apart from a few points already discussed. This suggests that the optimality model holds in these simple bilinear forms; that is, `cf02` generates loop nests that are optimal from both a theoretical and practical viewpoints. It is worthwhile noting how `auto` is not capable of selecting the proper synthesis in many of the test cases, especially as $nf \geq 2$. The fact that `cf02` is slower than `tens` at $[q = 4, p = 4, nf = 1]$ is explained by the memory threshold that prevents pre-evaluation. By just setting $T_H = L3_{size}$, `cf02` becomes $1.48\times$ faster than `tens`. This is probably due to the hardware prefetching being way more effective than in all other forms: the number of allocated temporaries is relatively small, and so is the number of memory access traces that need be tracked. However, as we already explained, in a parallel context this gain might diminish, since the L3 cache is shared by the cores of the CPU.

Helmholtz. Figure 8 shows that the performance improvements achieved by various code variants over a non-optimized implementation can be significant. We appreciate the fact that the relative order of the employed function spaces plays a key role in the code synthesis choice. Take, for example, the case $[q = 3, p = 1, nf = 2]$. The adoption of pre-evaluation makes a tremendous speed-up be achievable by `cf02` over the competitors. On the other hand, `auto` triggers quad (since one monomial in the form has two derivatives and two coefficients, which exceeds the threshold within which `tens` is selected), which, just like the other quadrature-based variants `ufls` and `cf01`, is sub-optimal. The effects of the working set threshold are in general positive. Moving to $T_H = L3_{size}$ makes `cf02` slower than `cf01` in many cases with $nf = 2$, the most significant one being a slow-down of $2.16\times$ at $[q = 2, p = 2]$.

Elasticity. The performance results for the elastic model are displayed in Figure 9. The fact that `auto` opts for `tens` when $nf = 1$ leads, generally, to sub-optimal execution times. This is different than in the case of the Helmholtz equation, in which the choice of `tens` was generally correct when $nf = 1$. Pre-evaluation is never triggered in `cf02`, because of either being estimated sub-optimal or exceeding T_H. In the former case, the performance improvements exhibited by `cf02` over cfO1 (and, in general, over all code variants) are due to the elimination of sharing and the avoidance of iteration over zero-valued blocks. The latter case occurs with $q = 4$ and $nf \in [0, 1]$. Running the same experiments with $T_H = L3_{size}$ resulted in a considerable improvement when $nf = 0$, with `cf02` becoming even faster than `tens`, while slow-downs characterized the cases of $nf = 1$ for all values of p. Our explanation for this is that when $[q = 4, nf = 0]$, T_H is exceeded by only 40%, while the cost function predicts a save in operation count of $5\times$. This suggests that our model could be refined to handle the cases in which the gain in operation count is so large that non-extreme variations in working set size should be considered negligible.

Hyperelasticity. In the hyperelastic model experiments, pre-evaluation is never triggered by `cf02`. This is a consequence of the form complexity, which makes the increase factor overwhelm any potential flop reduction. Another distinguishing aspect is the use of vector-valued function spaces, which requires a technique as in Section 5.2.3 to avoid wasteful operations over zero-valued entries; quad, `tens`, `ufls` and `cf02` employ different techniques. Results are displayed in Figure 10. `cf02` is the best alternative due to removing sharing from the multilinear loop nest and optimization of integration-dependent expressions. `ufls` performs generally well and exhibits significant speed-ups over non-optimized syntheses; this is a result of the effort in optimizing integration-dependent expressions.

7. CONCLUSIONS

With this research we have made a first step towards producing a theory and an automated system for the optimal synthesis of loop nests arising in finite element integration. The results are extremely encouraging, suggesting our model applies to a variety of contexts. We have discussed the conditions under which the model only leads to quasi-optimal loop nests. An open problem is understanding how to refine this model to include outer loops. This will probably require exploiting mathematical properties of differential operators. A second open problem is extending our methodology to classes of loops arising in spectral methods; here, the interaction with low level optimization will probably become stronger due to the typically larger working sets deriving from the use of high order function spaces. Lastly, we recall our work is publicly available and is already in use in the latest version of the Firedrake framework.

REFERENCES

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

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

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

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

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

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

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

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

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

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