A

Optimal 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 for arbitrary multilinear forms. Optimality is defined in terms of floating point operations required to compute a result. We establish a model under which optimality can systematically be reached and comment on the general applicability of this model. The model is integrated in a compiler that, given a representation of an integration routine, derives a loop nest in optimal form. Extensive performance evaluation proves the robustness of the model, 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. Optimal Finite Element Integration. *ACM Trans. Arch. & Code Opt.* V, N, Article A (January YYYY), 20 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

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

of derivatives, pre-multiplying functions, or polynomial order of the chosen function spaces, the operation count increases, thus making assembly covering a significant fraction of the overall runtime.

As demonstrated across a great deal of works, 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 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 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 ([**?**]), 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? With this paper, we aim to make a first step towards solving this problem.

We summarize the contributions of this research 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 an incremental, constructive proof that optimal implementations can be obtain by analyzing the monomials appearing in a form
— We automate the translation of forms into optimal loop nests by a compiler, COFFEE[1], the default optimization system in the Firedrake framework.
— We provide an unprecedented (for its extension) performance evaluation comparing several code transformation systems in a range of forms of increasing complexity. This is essential to demonstrate that our optimality model holds in practice.
— We comment on the cases in which our characterization of optimality becomes weaker, and provide an intuition of how we handle such forms
— We enrich the set of low-level transformations described in Luporini et al. [2015]. These make COFFEE a full-fledged system capable of decoupling form compilers from low level code generation.

2. PRELIMINARIES

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

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

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 a change of variables from reference coordinates X_0 to real coordinates $x = F_K(X_0)$ is necessary any time a new element is evaluated. The numerical integration routine based on quadrature over an element K can be expressed as follows

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

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

2.2. Tensor Contraction Mode

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

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

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

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

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

2.3. Qualitative Comparison

Depending on form and discretization, the relative performance of the two modes, in terms of the operation count, can vary even 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 more indicate 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. OPTIMALITY OF LOOP NESTS

In this section, we provide a characterization of loop nest optimality. In order to make the document self-contained, we start with reviewing basic compiler terminology.

Definition 1 (Perfect and imperfect loop nests)**.** *A loop nest is said to be perfect when non-loop statements appear only in the body of the innermost loop. If this condition does not hold, a loop nest is said imperfect.*

Definition 2 (Independent basic block)**.** *A sequence of statements that present no data dependencies compose an independent basic 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 the obvious loop dependencies are preserved. 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 a refinement that goes beyond the scope of this paper.

It is convenient to map mathematical properties to the loop nest level

Definition 3 (Linear loop)**.** *A loop L defining the iteration space I through the iteration variable i, or simply L_i, is linear if all expressions appearing in the body of L that use i to access some memory locations are linear functions over I.*

In particular, we are interested in the following class since it naturally arises from the math of finite element integration, reviewed in Section 2.

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.*

Note that perfect multilinear loop nests could actually be rooted in deeper, possibly imperfect loop nests. In fact, we will focus on this particular structure.

We introduce the fundamental notion of sharing.

Definition 5 (Sharing)**.** *A loop L_i presents sharing if it contains at least two expressions depending on i that are symbolically identical.*

Figure 1(a) shows an example of a trivial multilinear loop nest of arity $n = 2$ with sharing along dimension j.

```
for (j = 0; j < O; j++)
  for (i = 0; i < N; i++)
    a += B[j]*C[i] + B[j]*D[i]
```

(a) With sharing

```
for (i = 0; i < N; i++)
  T = C[i] + D[i]
  for (j = 0; j < O; j++)
    a += B[j]*T
```

(b) Optimal form

Fig. 1: A simple multilinear loop nest

We now have the ingredients to formulate a simple yet fundamental result.

Proposition 1. *Assume $LN = [L_{i_0}, L_{i_1}, ..., L_{i_{n-1}}]$ is a perfect multilinear loop nest with the body of $L_{i_{n-1}}$ being an independent basic block. Then sharing can always be eliminated from LN.*

Proof. The demonstration is by construction and exploits linearity. We want to transform LN into LN' such that there is no sharing in any $L_i \in LN'$. Starting from the innermost loop $L_{i_{n-1}}$, the expressions are "flattened" by expanding all products involving terms depending on $L_{i_{n-1}}$. Being now on the same level of the expression tree, such terms can be factorized. Due to linearity, each factored product only has one term depending on $L_{i_{n-1}}$, and such term is now unique in the expression. The other terms, independent of $L_{i_{n-1}}$, are, by definition, loop-invariant, and as such can be hoisted at the level of $L_{i_{n-2}}$. This procedure can be applied recursively up to L_{i_0}: multilinearity allows factorization at each level; perfectness ensures hoisting is safe. □

This result will be essential to prove that optimality, characterized as follows, can always be reached in this class of loops.

Definition 6 (Optimality of a multilinear loop nest)**.** *The synthesis of a multilinear loop nest is optimal if the amount of operations performed in the innermost loop is minimum.*

In other words, optimality implies there is no other synthesis able to further decrease the number of operations in the innermost loop. Note that this definition 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. In the following, we assume to operate in a CPU-bound regime, in which arithmetic-intensive expressions need be evaluated. This suits the context of finite element integration.

A second result follows.

Proposition 2. *A perfect multilinear loop nest LN with basic block independence can always be reduced to optimal form.*

Proof. By construction. Loop-dependent terms are logically grouped into n disjoint sets S_i, each S_i containing all terms depending on L_i. These sets are sorted in descending order based on their cardinality. We produce a new loop permutation that reflects the S_i ordering; that is, loops are placed such that, as going down the nest, L_i is characterized by less unique terms than L_{i-1}. Note the loop permutation is semantically correct because of perfectness. Starting from $L_{i_{n-1}}$, we can apply the sharing-removal procedure described in Proposition 1. This renders $L_{i_{n-1}}$ optimal. In particular, the number of operations is equal to $LN_{ops} = \#S_{i_{n-1}} + (\#S_{i_{n-1}} - 1)$, in which the second term represents the cost of summing the $\#S_{i_{n-1}}$ terms. □

For example, by applying the construction described in Proposition 2 to the code in Figure 1(a), we obtain the optimal form in Figure 1(b).

We now consider a wider class of loop nests in which perfectness and multilinearity apply to a sub-nest only. Consider the example in Figure 2.

```
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[e,j,k] += F(...)
```

Fig. 2: An imperfect loop nest, with one reduction loop (L_i), an inner perfect multilinear loop nest ($[L_j, L_k]$), and sharing potentially present in F

The imperfect nest $LN = [L_e, L_i, L_j, L_k]$ comprises a reduction loop L_i and a perfect doubly nested loop $[L_j, L_k]$, which we assume to be multilinear. The right hand side of the statement computing the multidimensional array `A` is a generic expression `F` including standard arithmetic operations such as addition and multiplication, possibly characterized by sharing. Here, one could think of searching for sub-expressions for which the reduction could be pre-evaluated, thus obtaining a decrease proportional to M in the operation count. This is exemplified through Figures 3(a) and 3(b) by fixing F in a simpler loop nest.

It is then obvious that reaching an optimal form LN' is, in the general case, a challenging issue, since a full exploration of the expression tree transformation space is

```
for (i = 0; i < M; i++)
  for (k = 0; k < O; k++)
    A[k] += B[i,k]*(C[i]*a + D[i]*b)
```

(a) With reduction

```
for (i = 0; i < M; i++)
  for (k = 0; k < O; k++)
    T1[k] += B[i,k]*C[i]
    T2[k] += B[i,k]*D[i]

for (k = 0; k < O; k++)
  A[k] = T1[k]*a + T2[k]*b
```

(b) Pre-evaluated reduction

Fig. 3: The effect of pre-evaluating reductions. Note the decrease in operation count grows with the loop nest depth.

necessary. Even though we assumed we could generate LN' through a suitable transformation algorithm, the following issues should be addressed

— as opposed to what happens with hoisting in perfect multilinear loop nests, the temporary variable size would be proportional to the number of non-reduction loops crossed (in the example, $N \cdot O$ 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 sub-expressions amenable to pre-evaluation could increase the arithmetic complexity within L_k (for example, expansion may increase the operation count). This could overwhelm any gain due to pre-evaluation.

We shall then refine our definition of optimality for this wider class of loop nests as follows

Definition 7 (Optimality of a loop nest)**.** *The synthesis of a loop nest is optimal if, under a set of memory constraints* C*, the total amount of operations performed in all innermost loops is minimum.*

Note how the definition contemplates the possibility for a nest to have multiple innermost loops. In fact, multiple sub-nests could be rooted in the outermost loop, either because part of the input or as a result of some transformations (e.g., see Figure 3(b)).

4. SYNTHESIS OF OPTIMAL LOOP NESTS IN FINITE ELEMENT INTEGRATION

In this section, we instantiate Definition 7 to our domain of interest, finite element integration. This will require reasoning 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 2. This loop nest is actually a simplified view of a typical bilinear form implementation. L_e represents iteration over the elements of a mesh; L_i derives from using numerical quadrature; $[L_j, L_k]$ implement the computation over test and trial functions. We deliberately omitted useless portions of code to not hinder readability and to avoid tying our discussion to specific forms.

We make the following observations. 1) $L >>> M, N, O$; 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 $L >>> M, N, O$ suggests we should be cautious about hoisting out of LN. Imagine LN is enclosed in a time stepping loop. One could think of exposing and then hoisting time-invariant sub-expressions. There are several kinds of time-invariant sub-expressions: for example, those depending on the geometry (i.e., depending on L_e) and those which do not (e.g., when only involving reference element terms). In the former case, code motion increases the working set by a factor L (unless adopting complex engineering solutions, like blocking non-affine loop nests, which are practically difficult to devise and maintain). The drop in number of operations could then be overwhelmed, from the runtime viewpoint, by a much larger memory pressure.

A second, more general thought 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 LN, requires temporary arrays of size $N \cdot O$; with certain discretizations, as detailed in Section 6.2, this can break the local memory threshold.

Based upon these considerations, we add two elements to the set of memory constraints C (see Definition 7)

(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 LN, 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 Cost

Definition 7 states that a necessary condition for a loop nest synthesis to be optimal is that the sum of the innermost loop operation costs 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 2, 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 LN, 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.

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 mono-

```
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

mial, pre-evaluation will decrease the operation count, then the corresponding subexpression 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 2. 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: Optimized Loop nest example

Before elaborating on the profitability of pre-evaluation, we need to discuss under which conditions this approach, based on a "local analysis" of monomials, is optimal.

Proposition 3. *Consider an expression comprising a set of monomials M. Let P be the set of pre-evaluated monomials, determined as described in Figure 4, and be $Z = M \setminus P$. 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 loop nest LN is optimal under memory constraints C, once sharing is removed.

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 ignore without compromises. 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 LN: $[L_e, L_j, L_k]$, 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 LN synthesis is $LN_{ops} = LN_{ops_1} + LN_{ops_2} = \sum_{\alpha}^{\#P} p_\alpha + I \sum_{\beta}^{\#Z} z_\beta$, where p_α and z_β represent the operation cost of monomials in P and Z, respectively, while I is the iteration space size of L_i. This can also be inferred from Figure 5.

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 LN_{ops}; B) not pre-evaluating any $P_Z : P_Z \subseteq P$ would increase LN_{ops}.

A) We prove that $LN'_{ops} = LN'_{ops_1} + LN'_{ops_2} > LN_{ops}$. It is rather obvious that $LN'_{ops_1} \geq LN_{ops_1}$ (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 $LN'_{ops_2} = LN_{ops_2}$, so our statement is true. If, on the other hand, at least one monomial does not share any terms, we obtain $LN'_{ops_2} < LN_{ops_2}$ or, equivalently, $LN'_{ops_2} = LN_{ops_2} - I \cdot \delta$. What we have to show now is that even by exposing more pre-evaluations, $LN'_{ops_1} - LN_{ops_1} = \gamma \geq I \cdot \delta$ 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 $LN'_{ops_2} > LN_{ops_2}$, being the cost function optimal due to assumption 1). Assumption 3) guarantees there can be no sharing within P_Z, which avoids subtle cases wherein 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 terms in the innermost loop, adding p to Z does not augment the operation count in our optimality model, so we can just avoid pre-evaluation.

□

4.3. A-Priori Operation Counting

It remains to tie one loose hand: the construction of the pre-evaluation cost function. We introduce the cost function θ such that $\theta : M \to \mathbb{N} \times \mathbb{N}$; that is, given a monomial, two natural numbers representing the optimal 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 2. A simple analysis suffices to obtain the cost of a sharing-free multilinear loop nest, namely LN^{sf}_{ops}. Assuming I to be the size of the L_i iteration space, we have that $\theta_{wo} = LN^{sf}_{ops} \cdot I$.

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: Optimized Loop nest example

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. In other words, the resulting loop nest would be characterized by sharing. 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 instead 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 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 bi-dimensional Poisson equation, after a suitable factorization, we 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 analysis described in Section 4 has been fully automated in COFFEE, the optimization system for local assembly used in Firedrake. In this section, we describe the

structure of the code generation system and we also comment on a set of low-level optimizations.

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 definition of optimality becomes weaker, since the 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 remove 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 to 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 transforma-

tions 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].

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

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

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, 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 0.03% of the test cases (we are not counting marginal differences), `cf02` 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 `cf02` in `Mass` and `Helmholtz` when $q = 1$. With $p \in [2, 3]$, `tens` slightly outperforms `cf02`. With $p = 4$, `tens` is quite faster than `cf02`. 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 repeated 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.

A second observation concerns 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.

Also note some other minor things.

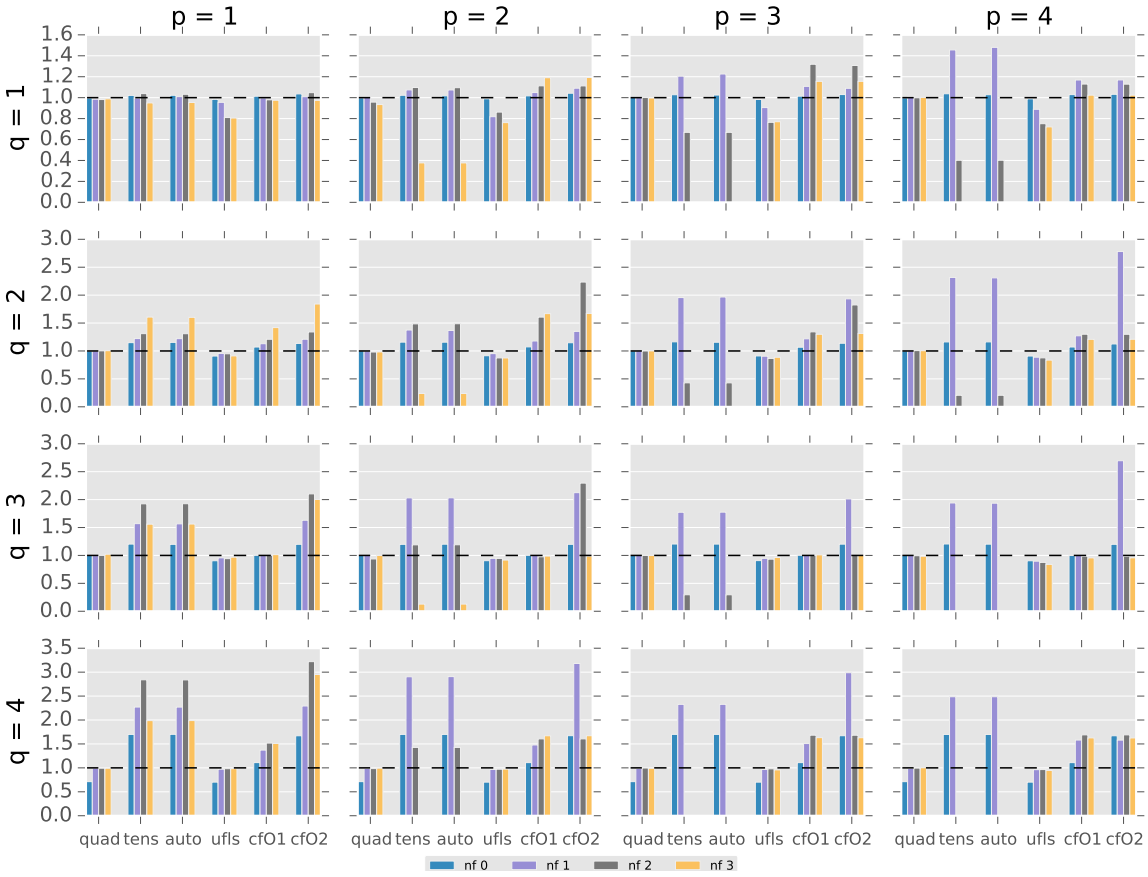

Fig. 7: Performance evaluation for the mass matrix. The bars represent speed up over the original, unoptimized code produced by the FEniCS Form Compiler.

— The lack of 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 all 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 characterizing 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,

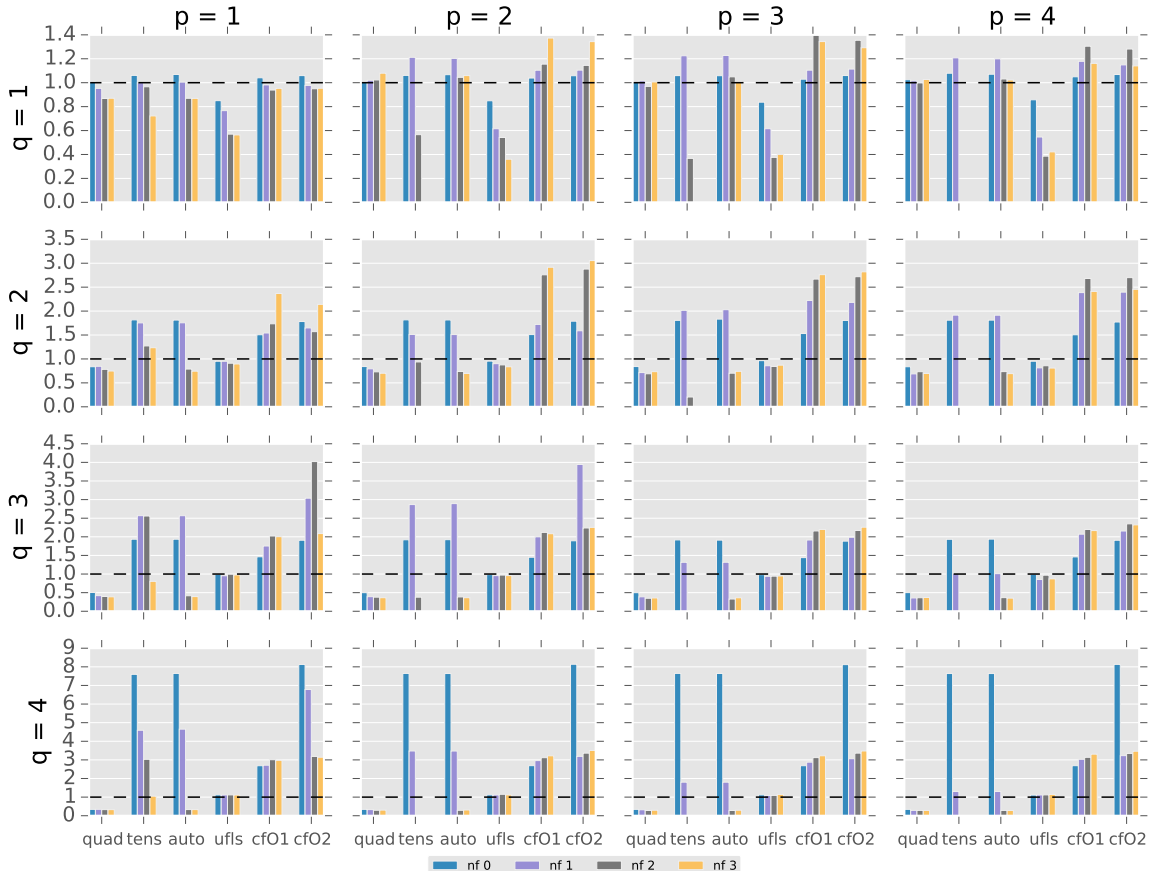

Fig. 8: Performance evaluation for the bilinear form of a Helmholtz equation. The bars represent speed up over the original, unoptimized code produced by the FEniCS Form Compiler.

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× 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

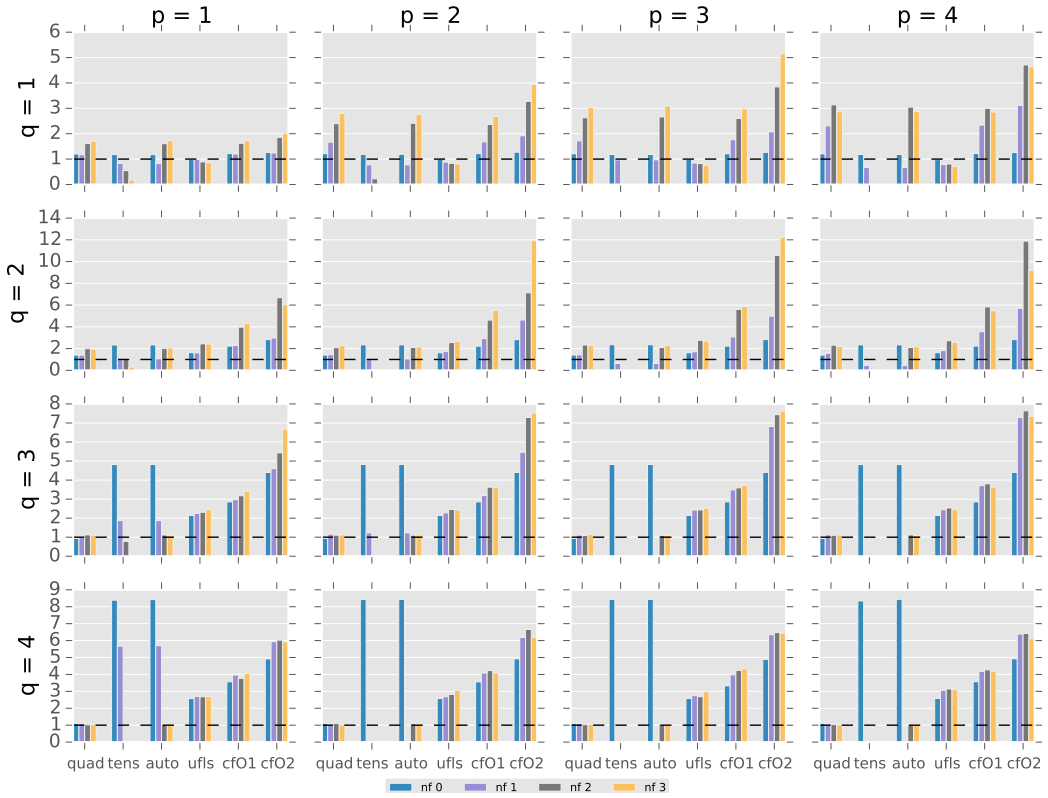

Fig. 9: Performance evaluation for the bilinear form arising in a elastic model. The bars represent speed up over the original, unoptimized code produced by the FEniCS Form Compiler.

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× 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 completely removing sharing and avoiding 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 roughly 40%, while the cost function predicts a save in operation count of 5×. This suggest that our model could be refined to handle the cases in which the gain in operation count is so large to make non-extreme variations in working set size negligible.

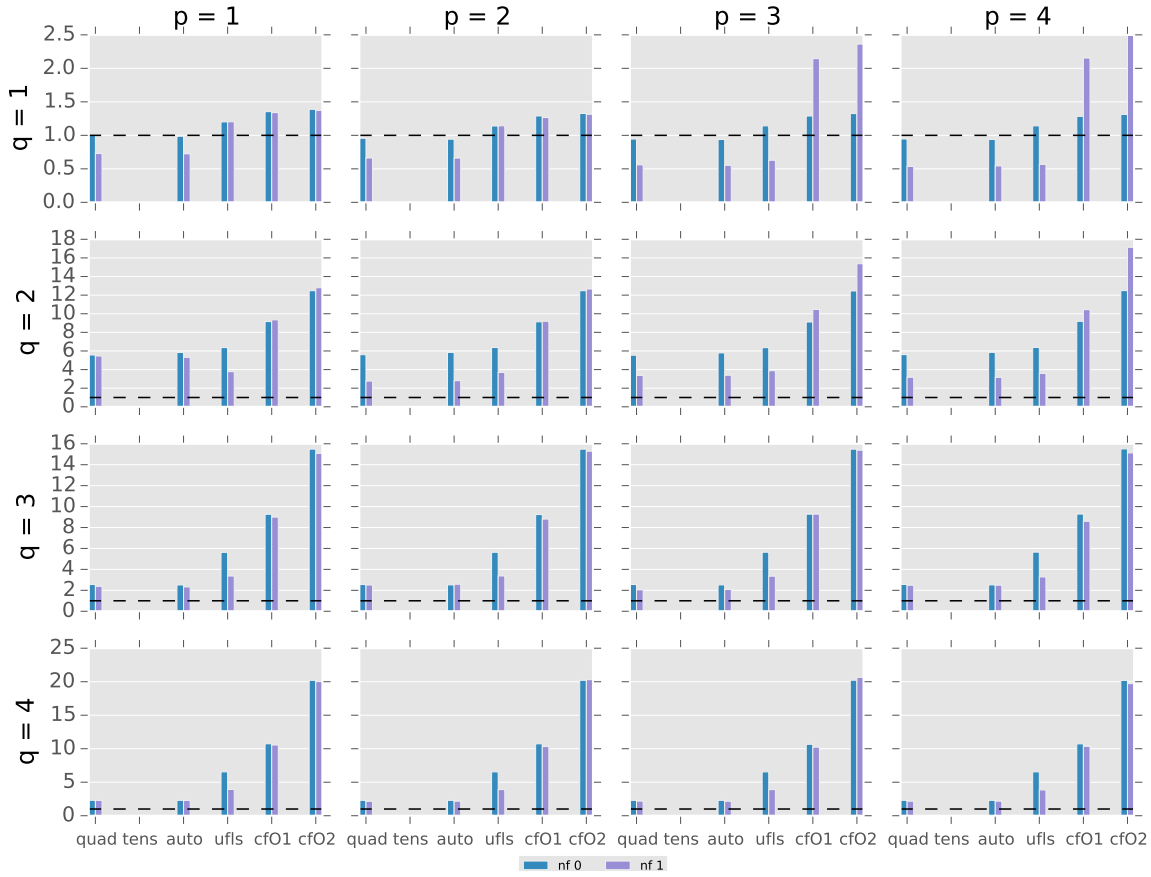

Fig. 10: Performance evaluation for the bilinear form arising in a hyperelastic model. The bars represent speed up over the original, unoptimized code produced by the FEniCS Form Compiler.

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 optimality model be-

comes weaker. An open problem is understanding how to refine this model to include outer loops. This will probably require exploiting 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

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

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).