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 such that, given a representation of an integration routine, a loop nest in optimal form can automatically be derived. Extensive performance evaluation proves the robustness of the model, showing considerable performance improvements over a number of alternative code generation systems. The effect of low-level optimization is also discussed.

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, which then 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 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 aggressive loop optimizations 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 transformation systems or synthesizers. In Olgaard and Wells [2010], it is shown how automated code generation can be leveraged to introduce domain-specific optimizations that a user should not be expected to write "by hand". In Kirby et al. [2005] 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. There is also an on-going effort to produce a novel form compiler, and therefore a novel code generation system for assembly operators, called UFLACS ([**?**]).

In spite of 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 in the direction of 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 synthesized by analyzing each monomial appearing in a form
— We implemented the theory in our compiler, COFFEE[1], which is the default optimization system in the Firedrake framework. Our implementation is general: we can apply our results to any forms expressible in Firedrake (and, theoretically, in FEniCS too).
— We provide an unprecedented (for extension) performance evaluation comparing several code transformation/synthesizers systems in a wide range of forms of increasing complexity, which is essential to demonstrate our optimality claim.
— We comment on the cases in which our characterization of optimality becomes weaker, and intuitions about how we deal with (i.e., optimize) them
— We enrich the set of low-level transformations described in Luporini et al. [2015]

2. PRELIMINARIES

We review finite element integration and possible implementations using 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 $U = V$ and $\{\phi_i\}$ 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 computed in the so called assembly phase. Then, in a subsequent phase, the linear system is solved, usually by means of an iterative method, and $\mathbf{u}$ is eventually evaluated.

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 the paper: this is about computing the contributions that an element in the discretized domain provide to the approximated solution of the equation. Global assembly, on the other hand, is the process of suitably "inserting" such contributions in A and b.

Without loss of generality, we illustrate local assembly in a concrete example; that is, 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 conveniently used to numerically evaluate A_{ij}^K. For convenience, a reference element K_0 and an affine mapping $F_K : K_0 \rightarrow K$ to any element $K \in T$ are introduced. This implies 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 representation over an element K

can be expressed as follows

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

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 basic mathematical properties we can rewrite the expression as

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

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 a temporary. The evaluation of the local tensor can then be abstracted as

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

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

2.3. Qualitative Comparison

Depending on the form being considered, the relative performance of the two modes, in terms of number of operations executed, 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 increasingly 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".

These two modes have been implemented in the FEniCS Form Compiler. In this compiler, a simple 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 is capable of selecting the optimal mode at the monomials level – no heuristics, no "global" choice of the mode for all monomials in the form – without affecting the cost of code generation.

3. OPTIMALITY OF LOOP NESTS

In this section, we provide a characterization of optimality for loop nests of various nature. 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.*

A straightforward property of perfect nests is that hoisting invariant expressions from the innermost loop to the preheader (i.e., the block that precedes the entry point of the nest) is a safe transformation. We will make use of this property.

Definition 2 (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 this work, we are particularly interested in the following class since it naturally arises from the math described in Section 2.

Definition 3 (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 4 (Sharing). *A loop L_i presents sharing if it contains at least two expressions depending on i that are symbolically identical.*

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

```
for (j = 0; j< O; j++)
  for (i = 0; i < N; i++)
    A[i,j] += B[j]*C[i]*b + B[j]*D[i]*c
```

Fig. 1: Multilinear loop nest with sharing

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

Proposition 1. *Sharing in a perfect multilinear loop nest $LN = [L_{i_0}, L_{i_1}, ..., L_{i_{n-1}}]$ can always be eliminated.*

Proof. The demonstration is by construction and exploits linearity. We want to transform LN into LN' such that there is no sharing is 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 Li_{n-1}. Being on the same level of the expression tree, such terms can then 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 always safe. $\square$

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

Definition 5 (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 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. By establishing a one-to-one mapping between set indices and loop indices, we produce a new loop permutation. The loop permutation is semantically correct because of perfectness. In this new order, loops are placed such that as going down the nest, L_i is characterized by less unique terms than L_{i-1}. 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, we obtain the optimal form in Figure 2.

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

Fig. 2: Multilinear loop nest in optimal form

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 3.

```
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. 3: Loop nest example

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. We observe that F might contain sub-expressions that depend on a subset of loops only and, as such, hoistable outside of LN as long as data dependencies are preserved (note that LN is imperfect). One could think of searching for sub-expressions independent

of L_e, for which the reduction could be pre-evaluated, thus obtaining a decrease proportional to M in the operation count. However, finding or exposing such reducible sub-expressions requires, in general, a full exploration of the expression tree transformation space. This is obviously challenging, and so is the problem of synthesizing an optimal LN' starting from LN. Even though we assumed we could generate LN' in non-exponential time, 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 run-time;
— the transformations exposing multi-invariant sub-expressions could require expansion and factorization. In terms of number of operations, the save originating from the elimination of the reduction loop could be overwhelmed by the increase in arithmetic complexity of L_k (e.g., expansion can increase the operation count, e.g. $A(B + C) = AB + AC$).

We then refine our definition of optimality for generic loop nests as follows

Definition 6 (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 result of suitable transformations.

4. SYNTHESIS OF OPTIMAL LOOP NESTS IN FINITE ELEMENT INTEGRATION

In this section, we instantiate Definition 6 in 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, which we reviewed in Section 2; and the (partly multilinear) loop nests implementing such forms.

Our point of departure is the example loop nest in Figure 3. 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 (e.g. matrix insertion) and to avoid tying our discussion to specific forms (e.g. F is unspecified).

From domain knowledge, 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 the iteration space of L_e is so larger than that of other loops suggests we should be cautious when hoisting out of LN. Imagine a time stepping loop L_t wraps LN. One could then think of identifying time-invariant sub-expressions that access both geometry and reference element terms and pre-evaluate them within L_t. Unless adopting complex engineering solutions (e.g. aggressive blocking), which are practically difficult to devise and maintain, this kind of code motion increases the working

set by $O(L)$. It is our opinion, therefore, that the drop in number of operations would be overwhelmed, from the run-time viewpoint, by the larger memory pressure.

A second, more general observation is that, for certain forms and discretizations, aggressive hoisting can make the working set exceed the size of "some level of local memory" (e.g. the last level of private cache on a conventional CPU, the shared memory on a GPU). We will provide precise details about this in the following sections. For the moment, and just as one of many possible examples, note that applying tensor contraction mode (see Section 2), which essentially means lifting code outside of L_e, requires a set of temporary arrays of size $N \cdot O$; with some discretizations, this can break the local memory threshold.

Based upon these considerations, we add two constraints to C (see Definition 6)

(1) The size of a temporary due to code motion cannot be bigger than the size of the multilinear loop nest iteration space.
(2) The total size of the hoisted temporaries cannot exceed a threshold T_H

A corollary of $C1$ is that hoisting expressions involving geometry terms outside of L_e becomes forbidden.

4.2. Minimizing the Operation Cost

Definition 6 states that a necessary condition for a loop nest synthesis to be optimal is that the number of operations in all innermost loops 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 6, 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 the input as in Figure 3, 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 algorithm for optimal loop nest synthesis is shown in Figure 4.

The point of departure consists of understanding the impact, as number of operations saved or introduced, of pre-evaluation. This is studied "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 2. The transformed loop nest is as in Figure 5.

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.

```
dissect the input expression into monomials
for each monomial M:
  θw = estimate operation count with pre-evaluation
  θwo = estimate operation count without pre-evaluation
  if θw < θwo and memory constraints satisfied:
    mark M as candidate for pre-evaluation
for each monomial M:
  if M does not share terms with M′, an unmarked monomial:
    extract M into a separate loop nest
    apply pre-evaluation to M
for each expression:
  remove sharing
```

Fig. 4: Intuition of the main algorithm

```
for (e = 0; e < L; e++)
  ...
  // Pre-evaluated tables
  ...
  // Loop nests for each dissected monomial
  for (j = 0; j < N; j++)
    for (k = 0; k< O; k++)
      A[e,j,k] += F(...)
    for (k = 0; k< O; k++)
      A[e,j,k] += G(...)
  ...
  // Loop nest for monomials for which run-time
  // integration (/i/ loop) 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

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 employed is optimal; that is, it predicts correctly whether pre-evaluation is profitable or not for a monomial
(2) pre-evaluating distinct monomials does not produce 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) How to create an optimal cost function is discussed in Section 4.3. 2) A pathological case due to symmetries in basis functions, which in practice rarely happens. 3) This could occur in complex forms with several monomials; for simplicity, we ignore this situation (otherwise, a "global" analysis of the monomials would be required).

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 notation by omitting all

shared loops when discussing operation counts. In particular, we use I to refer to the iteration space size of L_i. The operation count of what we are proving to be the optimal LN synthesis is, therefore, $LN_{ops} = LN_{ops_1} + LN_{ops_2} = \sum_{\alpha}^{\#P} p_\alpha + I \sum_{\beta}^{\#Z} z_\beta$, where p_α and z_β represent the operation cost of monomials in P and Z, respectively.

We start noting 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 symbols 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 symbols, 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, $LN'_{ops_1} \geq LN_{ops_1} + I \cdot \delta$ holds. This is indeed the case since we rely on assumption 2, which ensures the uniqueness of the pre-evaluated tables (i.e., the absence of sharing) and, therefore, the optimality of LN.

B) In absence of sharing, the statement is trivially true since we would have $LN'_{ops_2} > LN_{ops_2}$, with the cost function being optimal by assumption. Assumption 3 guarantees there can be no sharing within P_Z, which avoids subtle cases wherein pre-evaluation would result 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 "declassed" from P to Z: the rationale is that we would have to pay anyway the presence of the shared terms in the innermost loop due to z, so aggregating p in Z does not increase the operation count.   □

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 $\theta : M \to \mathbb{N} \times \mathbb{N}$ that, given a monomial, returns two natural numbers representing the optimal operation count without (θ_{wo}) and with (θ_w) pre-evaluation. 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. As explained in Proposition 2, a simple analysis suffices to obtain the cost of a sharing-free multilinear loop nest, namely MLN_{ops}. Then, assuming I to be the size of the L_i iteration space, we simply have that $\theta_{wo} = MLN_{ops} \cdot I$.

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

```
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_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 just a single coefficient, ι directly descends from the cost of expansion. In general, however, the calculation is less straightforward. Consider, for example, the case in which multiple coefficients originate from the same function space. Expansion would then lead to identical terms (i.e., identical pre-evaluated tables). Therefore, for a precise estimate of ι, we instead need to calculate the k-combinations with repetitions of n elements, with k being the arity of the multilinear loop nest and n the set of terms involved in the coefficient expansion.

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 employed. 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 the operators in the form; for convenience, in our implementation we instead project (by analysis of the expression tree) the output of expansion and factorization in the monomial.

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 FFC with quadrature and tensor modes, there are no separate trunks in COFFEE handling pre-evaluation, sharing, and code motion in general: all optimizations are expressed as composition of parametric "building-block" operations. This has several advantages. Firstly, extendibility: novel transformations – for instance, sum-factorization in spectral methods – could ideally 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 are parametric operations. For example, one could ask to factorize only constant or only linear terms, while hoisting can be limited by imposing a constraint on the size of the temporaries. Each of these operations is implemented by suitable 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 (e.g., see Olgaard and Wells [2010] and for coarser discretizations, the operation count within L_i could be comparable or, in some circumstances, outweigh that in 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 loops, we can still heuristically apply the same reasoning of Sections 3 and 4, particularly for removing sharing, to minimize the operational cost. 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 trivial code without performing any sort of transformation. 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 statements evaluating some temporaries. For each of this statements, we try to remove sharing by greedily applying factorization and code motion. In the COFFEE language, this process is expressed by simply concatenating five operators (i.e., five function calls named "factorize", "expand", etc.).

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. The blocking strategy consists of evaluating outer products by just using two vector register 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 relatively low number of 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 a separate loop nest.

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. In our loop nest, L_i is clearly an outer loop. In situations such as in Section 5.1.1, we may want to vectorize also along L_i. One way to achieve this is vector-promotion. This implies creating a clone loop of L_i in the preheader of the main loop nest, in which vector temporaries, instead of scalar ones, are evaluated.

5.2.3. Handling Block-sparse Tables. Consider a tabulated set of basis functions having quadrature points along the rows and functions along the 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 (and one of our goals is precisely to relieve it from the burden of smart code generation), there are several circumstances in which the tables can be block-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 general system that avoids useless iteration over these zero-entries 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 any 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 (rows, columns) of zero-valued entries, and if that slice is smaller than the vector length, that we do nothing (i.e., the loop nest is not transformed). Otherwise, we restructure the loop nest. This has several complicated implications, including loop fission and a memory offsetting usually dangerous for padding and data alignment, which are, however, beyond the scope of this paper. The implementation is based on symbolic execution: the loop nests are traversed and for each statement the location of zeros in each of the involved symbols is tracked. Arithmetic operators have a different impact on the tracking. For example, for each loop dimension, multiplication requires computing the set intersection of the zero-valued slices, 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 and 256KB L2 cache. 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 ([ufl 2014]).

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

— the FEniCS Form Compiler's built-in optimization systems, namely
   — optimized quadrature mode, quad
   — tensor mode, tens
   — automatic mode, auto (here, heuristics are used to infer the optimal mode for the form between quadrature and tensor)
— a novel back-end for the FEniCS Form Compiler, UFLACS, referred to as ufls, whose primary goals are improved code generation time and runtime
— the sole generalized loop-invariant code motion in COFFEE plus padding, cf01, as discussed in Luporini et al. [2015]
— the optimal multilinear loop nest synthesis plus padding and symbolic execution, cf02, which is the focus of this paper

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 a fair comparison, small patches (publicly available) were written to be able to use the aforementioned transformation systems in Firedrake; that is, all simulations are run through Firedrake. UFLACS and the FEniCS Form Compiler's optimization systems generate code suitable for FEniCS ([Logg et al. 2012]), which employs a data storage layout different than Firedrake. Our small patches fix this problem by producing code with a "correct" data storage layout.

Following the methodology (and the notation) used in [Olgaard and Wells 2010] and [Russell and Kelly 2013], 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 are instead

— the space of test, trial, and coefficient functions, Lagrange
— the mesh, tetrahedral with a total of 4374 elements

The upcoming Figures (7, 8, **??**, and 9) are "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 unmodified 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, general observation is that optimality, as we defined it, does not necessarily imply best achievable performance. There are two reasons for this. The first reason is that low-level optimizations can impact the actual runtime. Compare, as an example, tens and cf02 in Mass and Helmholtz when $q = 1$. With $p \in [2, 3]$, tens tend to slightly outperform cf02 even though pre-evaluation is triggered. With $p = 4$, tens is quite faster then cf02; in particular, in this case cf02 considers pre-evaluation unprofitable. The motivation is the aggressive unrolling performed in tens, which 1) allows to eliminate any operations involving zero-valued entries and 2) can drastically decrease the working set by not having to store entire temporaries. In COFFEE we never unroll loops to maximize the chances of autovectorization, which results in considerable speed ups throughout the majority of test cases. The second reason is a direct consequence of our axiomatic modeling of optimality: if we are in a regime as in Section 5.1.1, then other code generation strategies for the integration loop might result in larger performance improvements. This is the primary reason we are studying Hyperelasticity, a form characterized by extensive tensor algebra. Nevertheless, as we will see, we found cf02 to be the best scheme also in this circumstance.

A second observation concerns the triggering of pre-evaluation in cf02. The fact that a bar in cf02 matches the corresponding bar in cf01 means that pre-evaluation was found unprofitable. The trend – although there are quite a few exceptions – is that as nf increases, pre-evaluation tends to not be performed. This is a consequence of the larger increase factor, ι, which we defined in Section 4.3.

Also note some other minor things. The lack of improvements in all code variants with $[q = 1, p = 1]$ is due to the cost of matrix insertion, which outweighs that of local assembly. The slow downs experienced with quad should be imputable to the presence of indirection arrays in the generated code; when the tables are dense, these eventually break vectorization for a marginal gain due to avoiding iteration over isolated zero-valued columns. 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 in the case of Hyperelasticity)

Mass. Apart from a very small number of points in our exploration space, from analysis of Figure 7 we can infer how cf02 represents the best code variant among the tested. This suggests that our definition of optimality is consistent in what is the simplest possible bilinear form we can test. As we increase the complexity, by adding more coefficients or by varying the polynomial orders, cf02 is always capable of generating optimal loop nests. The biggest exception is $[q = 4, p = 4, nf = 1]$ in which the save on operations over zero-valued entries in tensor mode is considerably higher than the save in flops in cf01 and cf02.

Helmholtz. Figure 8 shows the performance improvements achieved by various code variants over the original code can be significant. It is worthwhile noting how auto is not capable of selecting the proper synthesis in most of the cases. One can also appreciate the evidence 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 all other variants. On the other hand, auto triggers quad (since one monomial in the form has two derivatives and two coefficients, which exceed the threshold within which tensor mode is applied), which, just like the other quadrature-based variants ufls and cf01, is sub-optimal.
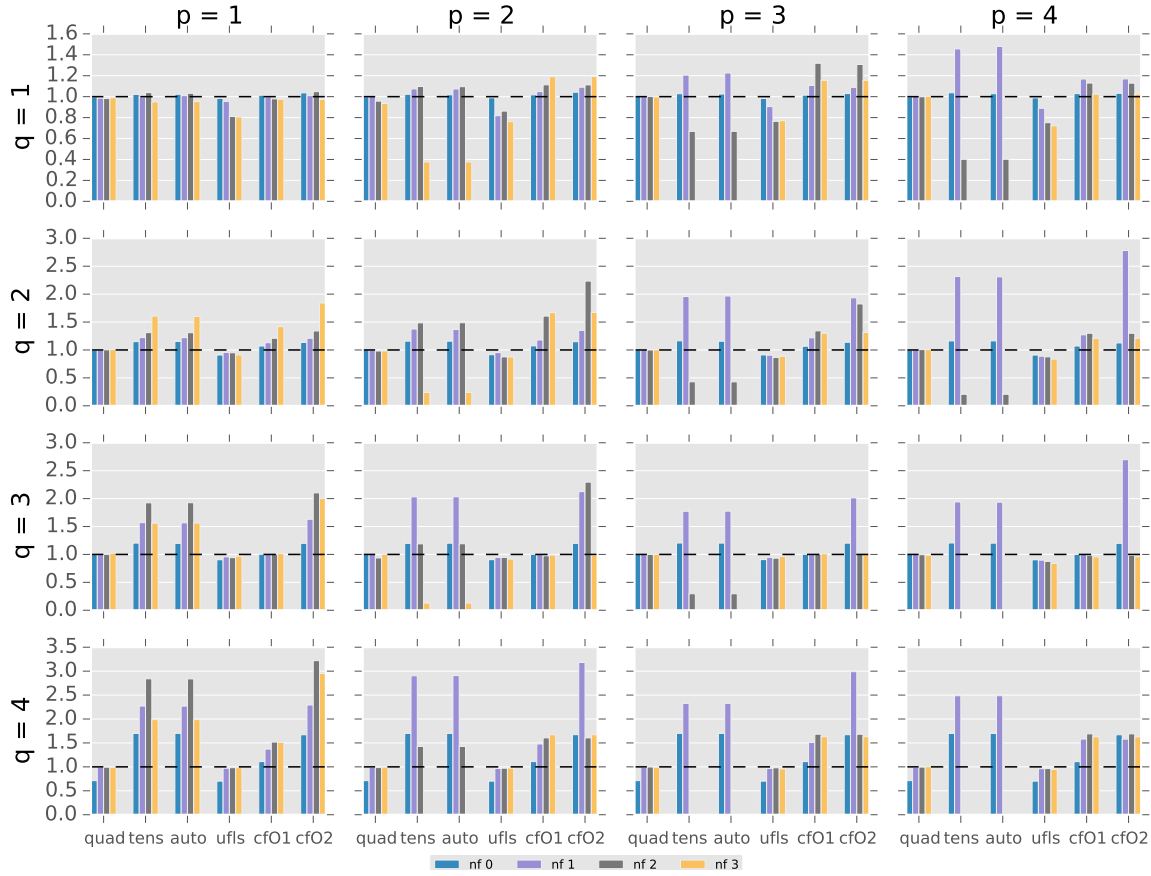
Elasticity.

Fig. 7: Performance evaluation for the mass matrix .

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 to this purpose. Results are displayed in Figure 9. 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

...

REFERENCES

2014. UFL forms. https://github.com/firedrakeproject/firedrake-bench/blob/experiments/forms/firedrake_forms.py. (2014).

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*
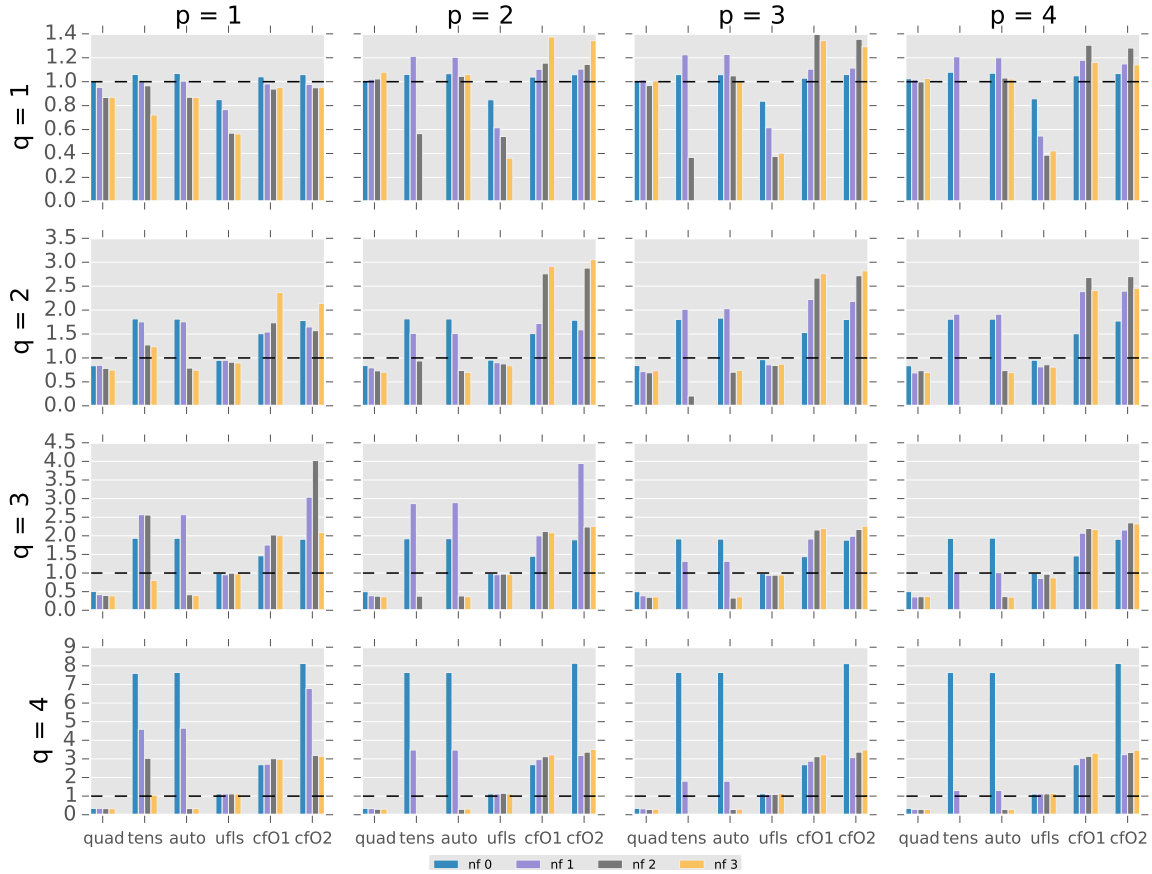
Fig. 8: Performance evaluation for the bilinear form of a Helmholtz equation.

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, Matthew Knepley, Anders Logg, and L. Ridgway Scott. 2005. Optimizing the Evaluation of Finite Element Matrices. *SIAM J. Sci. Comput.* 27, 3 (Oct. 2005), 741–758. DOI:http://dx.doi.org/10.1137/040607824

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).
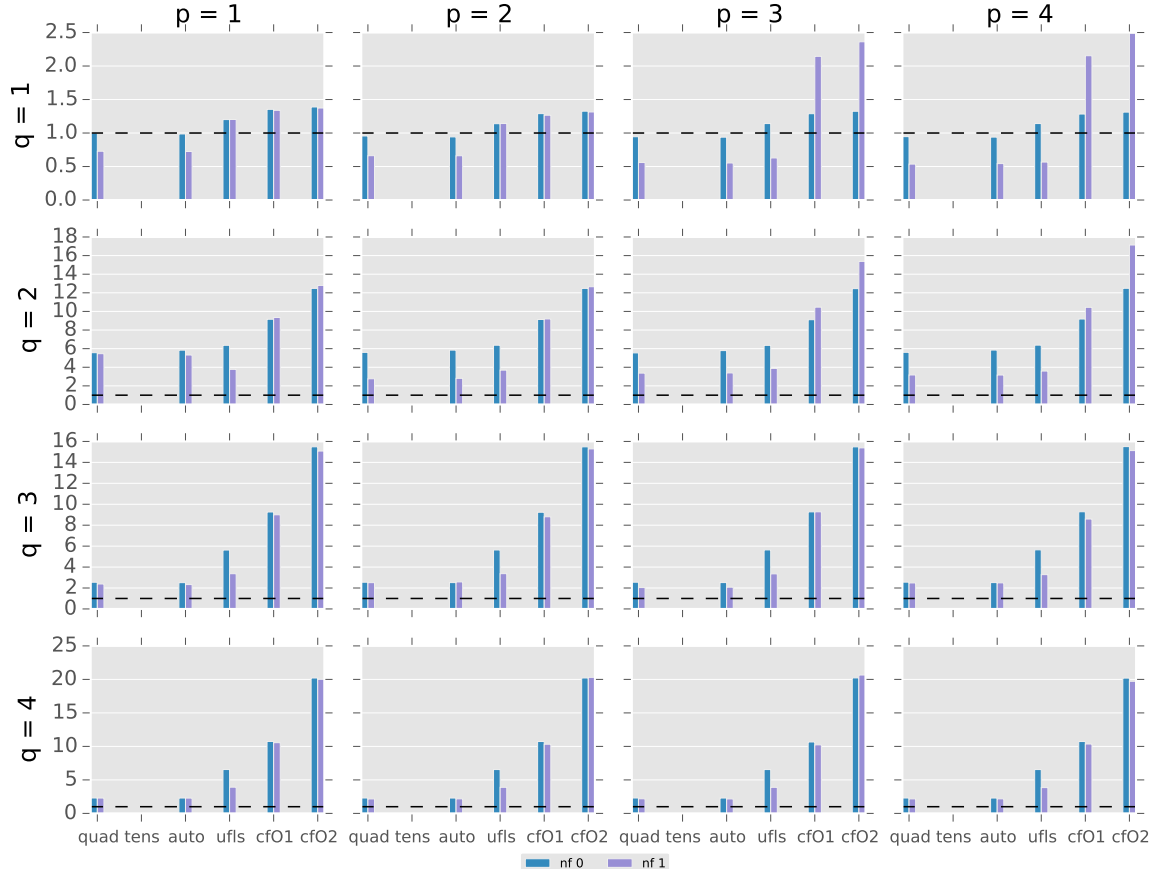
Fig. 9: Performance evaluation for the bilinear form arising in a Hyperelastic model