Optimal Finite Element Integration

Fabio Luporini, Imperial College London
David A. Ham, Imperial College London
Paul H.J. Kelly, Imperial College London

...

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, optimizations, SIMD vectorization

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

1. INTRODUCTION AND MOTIVATIONS

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 and then passed to a form compiler, which generates a representation of local assembly operations. These operations numerically evaluate problem-specific integrals in order to compute so called local matrices and vectors, which represent the contributions from each element in the discretized domain to the equation solution. Local assembly code must be high performance: as the complexity of a variational form increases, in terms of number of derivatives, pre-multiplying functions, and polynomial order of the chosen function spaces, the resulting assembly kernels become more and more computationally expensive, covering a significant fraction of the overall computation run-time.

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 Dr. Carlo Bertolli, 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

Producing high performance implementations is, however, non-trivial. The complexity of mathematical expressions involved in the numerical integration, which varies from problem to problem, and the small size of the loop nest in which such integral is computed obstruct the optimization process. Traditional vendor compilers, such as *GNU's* and *Intel's*, fail at exploiting the structure inherent assembly expressions. Polyhedral-model-based source-to-source compilers, for instance [Bondhugula et al. 2008], mainly apply aggressive loop optimizations, such as tiling, but these are not particularly helpful in our context. This lack of suitable optimizing tools has led to the development of a number of higher-level approaches to maximize the performance of local assembly kernels. In [Olgaard and Wells 2010], it is shown how automated code generation can be leveraged to introduce domain-specific optimizations, which a user cannot be expected to write "by hand". [Kirby et al. 2005] and [Russell and Kelly 2013] have studied, instead, different optimization techniques based on a mathematical reformulation of finite element integration. In [Luporini et al. 2014], we have made one step forward by showing that different forms, on different platforms, require distinct sets of transformations if close-to-peak performance must be reached, and that low-level, domain-aware code transformations are essential to maximize instruction-level parallelism and register locality. The problem of optimizing local assembly routines has been tackled recently also for GPU architectures, for instance in [Knepley and Terrel 2013], [Klöckner et al. 2009], and [Bana et al. 2014].

Our research has resulted in .... we build on our previous work [Luporini et al. 2014] ... and present a ... We argue that for complex, realistic forms, peak performance can be achieved only by ... ... also low-level optimisation ... ...also for the first time we provide a formal explanation of the math in terms of compiler theory

This is all implemented in COFFEE, which in turn is integrated with the Firedrake framework. We provide an extensive and unprecedented performance evaluation across a number of forms of increasing complexity, including some based on complex (hyperelasticity) models. We characterize our problems by varying polynomial order of the employed function spaces and number of pre-multiplying functions. To clearly distinguish the improvement achieved by this work, we will compare, for each test case, X sets of code variants: 1) unoptimized code, i.e. a local assembly routine as returned from the form compiler; 2) code optimized by FEniCS, i.e. the work in [Olgaard and Wells 2010]; 3) code optimized as described in [Luporini et al. 2014]; ....

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].

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 $F_K : K_0 \to K$ 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 representation over an element K can be expressed as follows

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

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 basic mathematical properties we can rewrite the expression as

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

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

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

in which the pre-evaluated "reference tensor" $A_{i_1 i_2 \alpha}$ and the cell-dependent "geometry tensor" G^α_K 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 characterize our definition of optimality and we describe the assumptions under which it holds.

In order to make the document self-contained, we start by reviewing basic compiler terminology.

Definition 1 (Perfect and imperfect loop nests)**.** *A loop nest is said to be perfect when code appears only in the body of the innermost loop. Conversely, the presence of code between any pair of loops in a nest renders the loop nest imperfect.*

A straightforward property of perfect nests is that hoisting of invariant expressions from the innermost loop to the preheader (i.e., the block that precedes the entry point of the nest) is always safe. We will later make extensive 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 over I.*

In this work, we are particularly interested in the following class since it naturally arises from the math described in 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 $l \in [0, n-1]$.*

Note that nothing prevents a perfect multilinear loop nest to be enclosed by an outer, possibly imperfect, nest. Indeed, this is the loop structure that will be at the centre of our study. We will later elaborate on this. First, we need to formulate a definition of optimality for perfect multilinear loop nests. To this purpose, it is convenient to introduce the 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.*

At this point, we can present a simple yet fundamental result.

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

Proof. The demonstration is by construction and exploits linearity. We want to transform N into N' such that sharing is absent at all levels of N'. Starting from the innermost loop $n-1$, the expressions are "flattened" by expanding all products involving any of the terms depending on i_{n-1}. Subsequently, such terms can be factorized. Due to linearity, in each factored product there is only one term depending on i_{n-1}. The factors that do not depend on i_{n-1} are, by definition, loop-invariant, and as such can be hoisted at the level of 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. □

Based on this proposition, we define optimality as follows.

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, this implies there is no other nest syntheses that could decrease the number of operations in the body of the innermost loop any further. Note that this definition of optimality 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" clearly makes no sense. In the following, we assume to operate in a CPU-bound regime, in which the body of loop nests are characterized by arithmetic-intensive expressions. This suits the context of finite element integration.

A second result follows.

Proposition 2. *An optimal synthesis for a perfect multilinear loop nest N can be found in ....*

Proof. By construction. We consider the body of the innermost loop $L_{i_{n-1}}$. Constant expressions can be hoisted outside of N and be replaced with suitable temporaries. Loop-dependent terms are grouped into n disjoint sets S_i, each S_i containing all terms depending on loop 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}. At this point, we can apply the

sharing-removal procedure described in Proposition 1. This renders $L_{i_{n-1}}$ no further improvable in the number of operations executed. In particular, the number of operations is equal to $\sharp S_{i_{n-1}} + (\sharp S_{i_{n-1}} - 1)$, in which the second term represents the cost of the summation. □

In practice, this result is too weak if we consider generic loop nests in which multilinearity applies to a subset of loops only and reductions are present. Consider the example in Figure 1.

```
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. 1: Loop nest example

The imperfect nest $N = [L_e, L_i, L_j, L_k]$ 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 `N`, provided that data dependencies are preserved (note that `N` is imperfect). One could think of pre-evaluating the reduction, thus obtaining a decrease proportional to `M` in the number of operations executed. However, finding or exposing such sub-expressions is, in general, challenging. Further, even though we assumed we could do so, the following issues should be addressed

— as opposed to what happens with hoisting in perfect multilinear loop nests, in this case the size of the temporaries would be proportional to the number of non-reduction loops (in the example, `N·O` for `ijk` sub-expressions, `L·N·O` for `eijk` ones). This might shift the loop nest from a CPU-bound to a memory-bound regime, which might be counter-productive for actual runtime;
— the process by which multi-invariant sub-expressions are exposed could require transformations like expansion and factorization. Consequently, the save originating from the elimination of the reduction loop could be overwhelmed by a potentially larger increase in the number of operations in the body of L_k (for example, consider how expansion tends to increase operations).

We then review 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 amount of operations performed in all innermost loops of the nest is minimum.*

Clearly, this definition remains vague until instantiated in a certain model. Once the model is fixed, we can argue about the meaning of `C` and try to produce an algorithm to reach optimality. This is tackled in the next section.

4. SYNTHESIS OF OPTIMAL LOOP NESTS IN FINITE ELEMENT INTEGRATION

In this section, we instantiate our definition of loop optimality in the domain of finite element integration. This requires elaborating on the junction between 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 1. This loop nest is a simplified view of a typical bilinear form implementation. The L_e loop represents iteration over the elements of a mesh; the L_i loop derives from using numerical quadrature; the perfect loop nest $[L_j, L_k]$ implements the computation over the test and trial functions' degrees of freedom. We deliberately omitted useless portions of code to not hinder readability (e.g. matrix insertion) and to avoid tying our discussing to a specific form (e.g. `F` is unspecified).

Before expanding on optimality, we make a few observations about properties of our domain. 1) We know that $L >>> M, N, O$; that is, even with high order basis functions, the number of elements is typically order of magnitude larger than both the number of quadrature points and degrees of freedom. 2) The loop nest enclosed by the integration loop is always 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 that we should be cautious when hoisting outside of the loop nest N. Imagine a time stepping loop L_t wraps N. Theoretically, one could think of identifying time-invariant sub-expressions that access both geometry and reference element terms, and hoist them out of L_e. 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 a factor L. It is our opinion, therefore, that the decrease in the number of operations would be completely overwhelmed by the bigger 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 temporaries of size equal to $N \cdot O$; depending on N, O and the cardinality of such set (or, equivalently, "the rank and size of the geometry tensor"), it is not so unlikely to eventually break the local memory threshold.

Based upon these considerations, we impose two memory constraints

— The size of a temporary due to code motion cannot be bigger than the size of the multilinear loop nest iteration space ($N \cdot O$ for the bilinear form in the example). A corollary is that hoisting of expressions involving geometry terms outside of L_e is forbidden.
— The total size of the hoisted temporaries cannot exceed a threshold T_H

4.2. Operation Minimization under Memory Constraints

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 multilinear loops does not suffice for nest optimality. In fact, we wonder whether the reduction imposed by L_i could be pre-evaluated outside of N, as already suggested in 6, thus potentially 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 1, we can always dissect `F` into distinct sub-expressions (the monomials). Each sub-expression is then factorized so as to split constant from $[L_i, L_j, L_k]$-dependent terms, the latter ones are hoisted outside of N, and finally pre-evaluated. 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 intuition of the algorithm to synthesize an optimal loop nest is shown in Figure 2.

```
M = dissect the input expression into monomials
for each monomial in M:
  C = estimate ops after pre-evaluation
  NC = estimate ops without pre-evaluation
  if C < NC and memory constraints satisfied:
    extract the monomial into a separate loop nest
    apply pre-evaluation
for each expression:
  remove sharing
```

Fig. 2: Intuition of the main algorithm

The point of departure consists of understanding the impact, as number of operations performed (ops, in the following), of pre-evaluation. This is studied "locally"; that is, for each monomial, in isolation. We will later discuss the optimality of this approach and how we can determine ops exploiting simple mathematical properties.

If we estimate that, for a given monomial, pre-evaluation will decrease the workload, 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 all resulting expressions by applying a procedure as described in Proposition 2.

The transformed, operation-minimum loop nest is as in Figure 3.

Before elaborating on how we determine C and NC of Figure 2, we introduce the following proposition about the optimality of this approach.

Proposition 3. *Consider an expression comprising a set of monomials M. Let P be the set of all monomials for which it has been estimated that pre-evaluation decreases their operation count. Then, if pre-evaluation is applied to all of the monomials in P, the resulting loop nest, enclosing an expression which is the sum of all monomials in $Z = M \setminus P$, is optimal under memory constraints C once sharing is removed.*

Proof. We start noting that C ensures the only hoisting operations allowed are those inherent to both pre-evaluation and sharing. This implies that if we 1) optimally decide what should be pre-evaluated and 2) remove sharing, no better synthesis can be found. We then need to prove that following: A) ignoring the prediction cost and pre-evaluating $Z_P : Z_P \subseteq Z$ can only increase the operation count; B) ignoring the

```
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. 3: Optimized Loop nest example

prediction code and not pre-evaluating $P_Z : P_Z \subseteq P$ can only increase the operation count. That is, local analysis of monomials leads to an optimal synthesis.

We observe that the cost of the loop nest when pre-evaluating only all monomials P is ....

A) If there is no sharing between terms in Z_P and $Z \setminus Z_P$, then the statement is trivially true. Conversely, if present, sharing can be removed. However, the operation count would still be larger than when pre-evaluating Z_P. In fact, the cost of the loop nest would be $Z_P^{pre} + P + \sharp L_i \cdot Z \setminus Z_P$, which is greater than ..., since $Z_P^{pre} > \sharp L_i \cdot Z_P$.

B) ...

□

We now only need to tie our loose hand...

4.3. Heuristic Optimization of Integration-dependent Expressions

5. AUTOMATED CODE GENERATION

— Deficiencies of previous approaches
— Building block operations in coffee: 1) eliminating reuse, 2) insights on the various tree algorithms employed

6. LOW-LEVEL OPTIMIZATION

...

6.1. Avoiding Iteration over Zero-valued Blocks by Symbolic Execution

6.2. Padding and Data Alignment

Padding and data alignment as described in [Luporini et al. 2014] must be refined for the case in which computation over zero-valued columns is avoided. We recall effective SIMD (auto-)vectorization can be achieved only if the innermost loop size is a multiple of the vector register length, in which case the compiler needs not to introduce a remainder scalar loop. In the case of an AVX instruction set, for example, we want to round the size of the innermost loops to the closest multiple of 4 (AVX registers can fit up to four double-precision floats). This can be done provided that arrays are padded, if necessary. Moreover, loads and stores instructions are efficient only if their addresses are suitably aligned to cache boundaries; this is obtainable by enforcing the base address of each padded array to be a multiple of the vector length.

Consider again the code in Figure **??**. The arrays in the loop nest `[j1,k1]` can be padded and the right bound of loop `k1` can be safely increased to 8: eventually, values computed in the region `M[0:3][6:8]` will be discarded. Then, by explicitly aligning arrays and using suitable pragmas (e.g. `#pragma simd` for the Intel compiler), effective SIMD auto-vectorization can be obtained for this loop nest.

There are some complications in the case of loops `[j0,k0]`. Here, increasing the loop bound to 4 is still safe, assuming that both `T1` and `A` are padded with zero-valued entries, but it has no effect: the starting addresses of the load instructions would be `T1[3]` and `A[i][3]`, which are not aligned. One solution is to start iterating from the closest index that would ensure data alignment: in this specific case, $k0 = 0$. However, this would imply losing (partially in general, totally for this loop nest) the effect of the zero-avoidance transformation. Another possibility is to attain to non-aligned accesses. COFFEE can generate code for both situations, so we leave the autotuner, described in Section **??**, in charge of determining the optimal transformation.

Note that in some circumstances the previous solution cannot be applied, since extra iterations might end up accessing non-zero entries in the local element matrix `M`. In such a situation, there is no possibility of recovering data alignment.

6.3. Vector-promotion of Integration Quantities

....

7. PERFORMANCE EVALUATION

7.1. Experimental Setup

Experiments were run on a single core of an Intel architecture, a Sandy Bridge I7-2600 CPU, running at 3.4GHz, 32KB L1 cache and 256KB L2 cache. The Intel `icc 14.2` compiler was used. The compilation flags used were `-O3, -xHost, -xAVX, -ip`.

We analyze the run-time performance of four fundamental real problems, which comprise the differential operators that are most common in finite element methods. In particular, our study includes problems based upon the Helmholtz and Poisson equations, as well as elasticity- and hyperelasticity-like forms. The Unified Form Language [Alnæs et al. 2014] specification for these forms, which is the domain specific language that both Firedrake and FEniCS use to express weak variational form, is available at [ufl 2014].

We evaluate the *speed ups* achieved by three sets of optimizations over the original code; that is, the code generated by the FEniCS Form Compiler when no optimizations are applied. In particular, we analyze the impact of the FEniCS Form Compiler's built-in optimizations (henceforth `ffc`), the impact of COFFEE's transformations as presented in [Luporini et al. 2014] (referred to as `fix`, in the following), and the effect of Expression Rewriting and Code Specialization as described in this work (henceforth `auto`, to denote the use of autotuning as described in Section **??**). The `auto` values do not include the autotuner cost, which is commented aside in Section **??**.

The values that we report include the cost of local assembly as well as the cost of matrix insertion. However, the unstructured mesh has been made small enough to fit the L3 cache, so as to minimize the "noise" due to any operations that are not part of the element matrix evaluation itself. However, it has been reiterated over and over (e.g. [Olgaard and Wells 2010]) that as the complexity of a form increseas, the cost of local assembly becomes dominant. All codes were executed in the context of the Firedrake framework.

We vary several aspects of each form, which follows the approach and the notation of [Olgaard and Wells 2010] and [Russell and Kelly 2013]

— The polynomial order of basis functions, $q \in \{1, 2, 3, 4\}$

...

— The polynomial order of coefficient (or "pre-multiplying") functions, $p \in \{1, 2, 3, 4\}$
— The number of coefficient functions $nf \in \{0, 1, 2, 3\}$

On the other hand, other aspects are fixed

— The space of both basis and coefficient functions is Lagrange
— The mesh is three-dimensional, made of tetrahedrons, for a total of 4374 cells

Figures **??**, **??**, **??**, and **??**, which will be deeply commented in the next section, must be read as "plots, or grids, of plots". Each grid (figure) has two logical axes: p varies along the horizontal axis, while q varies along the vertical axis. The top-left plot in a grid shows speed ups for $[q = 1, p = 1]$; the plot on its right does the same for $[q = 1, p = 2]$, and so on. The diagonal of the grid shows plots for which basis and coefficient functions have same polynomial order, that is $q = p$. Therefore, a grid can be read in many different ways, which allows us to make structured considerations on the effect of the various optimizations.

A plot reports speed-ups over non-optimized FEniCS-Form-Compiler-generated code. There are three groups of bars, each group referring to a particular version of the code (`ffc, fix, auto`). 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.

7.2. Performance of Forms

The four chosen forms allow us to perform an in-depth evaluation of different classes of optimizations for local assembly. We limit ourselves to analyzing the cost of computing element matrices, although all of the techniques presented in this paper are immediately extendible to the evaluation of local vectors. As anticipated, in the following we comment speed ups of `ffc`, `fix`, and `auto` over the non-optimized, FEniCS-Form-Compiler-generated code.

We first comment on results of general applicability. By looking at the various figures, we note there is a trend in COFFEE's optimizations to become more and more effective as q, p, and nf increase. This is because most of the transformations applied aim at optimizing for arithmetic intensity and SIMD vectorization, which obviously have a strong impact when arrays and iteration spaces are large. The corner cases of this phenomenon are indeed $[q = 1, p = 1]$ and $[q = 4, p = 4]$. We also observe how `auto`, in almost all scenarios, outperforms all of the other variants. In particular, it is not a surprise that `auto` is faster than `fix`, since `fix` is one of the autotuner's tested variants, as explained in Section **??**. This proves the quality of the work presented in this paper, which shows significant advances over [Luporini et al. 2014]. The reasons for which `auto` exceeds both original code and `ffc` are discussed for each specific problem next. Also, details on the "optimal" code variant determined by autotuning are given in Section **??**.

Helmholtz. The results for the Helmholtz problem are provided in Figure **??**. We observe that `ffc` slows the code down, especially for $q \geq 3$. This is a consequence of using indirection arrays in the generated code that, as explained in Section 6.1, prevent, among the other compiler optimizations, SIMD auto-vectorization. The `auto` version results in minimal performance improvements over `fix` when $nf = 0$, unless $q = 4$. This is due to the fact that if the loop over quadrature points is relatively small, then close-to-peak performance is obtainable through basic expression rewriting and code specialization; in this circumstance, generalized loop-invariant code motion and padding plus data alignment. The trend changes dramatically as nf and q increase: a more ample spectrum of transformations must be considered to find the optimal local

...
...
...

assembly implementation. We will provide details about the selected transformations in the next section.

Elasticity. Figure **??** illustrates results for the Elasticity problem. This form uses a vector-valued space for the basis functions, so here transformations avoiding computation over zero-valued columns are of key importance. The `ffc` set of optimizations leads to notable improvements over the original code at $q = 1$. The use of inderection arrays allows to phisically eliminate zero-valued columns at code generation time; as a consequence, different tabulated basis functions are merged into a single array. Therefore, despite the execution being purely scalar because of indirection arrays, the reduction in arithmetic intensity and register pressure imply improvement in performance. Nevertheless, `auto` remains in general the best choice, with gains over `ffc` that are wider as p and nf increase.

For $q \geq 2$, in `ffc` the lack of SIMD vectorization counterbalances the decrease in the number of floating point operations, leading to speed ups over the original code that only occasionally exceed $1\times$. On the other hand, the successful application of the zero-avoidance optimization while preserving code specialization plays a key role for `auto`, resulting in much higher performance code especially at $q = 2$ and $q = 3$.

It is worth noting that speed ups of `auto` over `fix` decrease at $q = 4$, particularly for low values of p. As we will discuss in Section **??**, this is because at $q = 4$ the vector-register tiling transformation (in combination with loop unroll-and-jam) leads to the highest performance. In principle, vector-register tiling can be used in combination to the zero-avoidance technique; however, due to mere technical limitations, this is currently not supported in COFFEE. Once solved, we expect much higher speed ups in the $q = 4$ regime as well.

Poisson. In Figure **??** we report speed ups of `ffc`, `fix`, and `auto` over the original code for the Poisson form. We note that, as a general trend, `ffc` exhibits drops in performance as nf increases, notably when $nf = 3$, for any values of q and p. This is a consequence of the inherent complexity of the generated code. The way `ffc` performs loop-invariant code motion leads to the pre-computation of integration-dependent terms at the level of the integration loop, which are characterized by higher arithmetic intensity and redundant computation as nf increases. Moreover, the absence of vectorization is another limiting factor.

The `auto` variant generally shows the best performance. Significant improvements over `fix` are also achieved, notably as q, p and nf increase. As clarified in the next section, this is always due to a more aggressive expression rewriting in combination with the zero-avoidance technique.

Hyperelasticity. Speed ups for the hyperelasticity form are shown in Figure **??**. Experiments for $nf \geq 2$ could not be executed because of FEniCS-Form-Compiler's technical limitations.

For `auto`, massive speed ups for $q \geq 2$ are to be ascribed to aggressive and successful expression writing. Hyperelasticity problems are really compute-intensive, with thousands of operations being performed, so reductions in redundant and useless computation are crucial. Complex forms like hyperelasticity would benefit from further "specialized" optimizations: for example, it is a known technical limitation of COFFEE that, in some circumstances, less temporaries could (should) be generated and that hoisted code could (should) be suitably distributed over different loops to minimize register pressure (e.g. COFFEE could apply loop fission for obtaining significantly better

register usage). We expect to obtain considerably faster code once such optimizations will be incorporated.

In the regime $q \geq 2$ and $nf = 1$, peformance improvements are less pronounced moving from $p = 1$ to $p = 2$, although still significant; in particular, we notice a drop at $p = 2$, followed by a raise up to $p = 4$. It is worth observing that this effect is common to all sets of optimizations. The hypothesis is that this is due to the way coefficient functions are evaluated at quadrature points (identical in all configurations), which cannot be easily vectorized unless a change in storage layout and loops order is implemented in the code (abstract syntax tree) generator on top of COFFEE.

8. CONCLUSIONS

...

REFERENCES

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

M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. 2014. Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans Math Software* 40, 2, Article 9 (2014), 9:1–9:37 pages. DOI:http://dx.doi.org/10.1145/2566630

Krzysztof Bana, Przemyslaw Plaszewski, and Pawel Maciol. 2014. Numerical Integration on GPUs for Higher Order Finite Elements. *Comput. Math. Appl.* 67, 6 (April 2014), 1319–1344. DOI:http://dx.doi.org/10.1016/j.camwa.2014.01.021

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, 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

A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. 2009. Nodal Discontinuous Galerkin Methods on Graphics Processors. *J. Comput. Phys.* 228, 21 (Nov. 2009), 7863–7882. DOI:http://dx.doi.org/10.1016/j.jcp.2009.06.041

Matthew G. Knepley and Andy R. Terrel. 2013. Finite Element Integration on GPUs. *ACM Trans. Math. Softw.* 39, 2, Article 10 (Feb. 2013), 13 pages. DOI:http://dx.doi.org/10.1145/2427023.2427027

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. 2014. COFFEE: an Optimizing Compiler for Finite Element Local Assembly. (2014).

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).