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), 10 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 ...

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 summarize the basic concepts sustaining the finite element method following the notation 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 called bilinear and linear form, respectively. 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 Mode

...

3. OPTIMALITY OF MULTILINEAR LOOP NESTS

We need to define:

— multilinear loop nests, static loop nests; (note: the loop over the elements could also be considered static, if mesh didn't move...)
— concept of reuse in a loop;
— prove that reuse can always be eliminated from the innermost loop of multilinear loop nests.
— definition of optimality
— insights that optimisation of loops outside of the multilinear nest can be tackled greedily and heuristically

Need to show examples.

4. AUTOMATED CODE GENERATION OF OPTIMAL MULTILINEAR LOOP NESTS

— Deficiencies of previous approaches
— Building block operations in coffee: 1) eliminating reuse, 2) insights on the various tree algorithms employed
— cost model for deciding what to do
— constrained increase in the working set size
— heuristic optimisation of constant and quadrature dependent expressions

5. LOW-LEVEL OPTIMIZATION

...

5.1. Avoiding Iteration on Zero-valued Blocks by Symbolic Execution

Operations over blocks of zero-valued entries in tabulated (derivatives of) basis functions can be skipped. Zero-valued columns arise, for example, when taking derivatives on a reference element, or when employing vector function spaces. 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]]`, where `A` is a tabulated basis function and `B` a map from loop iterations to non-zero columns in `A`), was described and tested in `FFC`. We will evaluate our approach and compare it to this pioneering work in Section 6. Our strategy differs by avoiding indirection arrays in the generated code.

Consider Figure 1(a), an enriched version of the Burgers excerpt in Figure **??**. The array `D` represents a tabulated derivative of a basis function at the various quadrature points. There are four zero-valued columns. Any multiplications or additions along these columns could (should) be skipped to avoid irrelevant floating point operations. The solution adopted in [Olgaard and Wells 2010] is not to generate the zero-valued columns (i.e. to generate a dense 6×2 array), to reduce the size of the iteration space over test and trial functions (from 6 to 2) and to use an indirection array (e.g. $ind = \{3, 5\}$) to update the right entries in the element tensor A.

Our approach is based on using domain knowledge and symbolic execution. We discern the origin of zero-valued columns: for example, those due to taking derivatives on the reference element from those inherent to using mixed (vector) elements. In the running Burgers example, the use of vector function spaces require the generation of a zero-block (columns 0, 1, 2 in the array `D`) to correctly evaluate the local element matrix while iterating along the space of test and trial functions. The two key observations are that 1) the number of zero-valued columns caused by using vector function spaces is, often, much larger then that due to derivatives, and 2) such columns are contiguous in memory. Based on this observation, we aim at avoiding iteration only along the block of zero-valued columns induced by mixed (vector) elements.

We achieve our goal by means of symbolic execution. The Expression Rewriter expects some indication about the location of the zero-valued columns induced by mixed (vector) function spaces, for each tabulated basis function. This indication comes ei-

```
void burgers(double M[6][6],
             double **coordinates,
             double **coeff0) {
  // Calculate determinant of jacobian
  // (det) using the coordinates field

  // Define tabulation of basis functions
  // (and their derivatives) arrays
  ...
  @coffee mfs[3, 5]
  static const double D[6][6] =
     {{0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0}};
  ...
  for (int i=0; i<6; ++i) {
    ...
    double T1[6], T2[6];
    for (int r=0; r<6; ++r) {
      T1[r] = a*A[i][r]+b*B[i][r];
      T2[r] = d*D[i][k]+e*E[i][k];
    }
    for (int j=0; j<6; ++j)
      for (int k=0; k<6; ++k)
        M[j][k] += (T1[j], T2[k], A[i][j], ...)
  }
}
```

(a) Original (simplified) code. Note the annotation over the definition of the tabulated basis function `D`, which is used to identify the presence of zero-valued columns

```
void burgers(double M[6][6],
             double **coordinates,
             double **coeff0) {
  // Calculate determinant of jacobian (det)
  // using the coordinates field

  // Define tabulation of basis functions
  // (and their derivatives) arrays
  ...
  @coffee mfs[3, 5]
  static const double D[6][6] =
     {{0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0}};
  ...
  for (int i=0; i<6; ++i) {
    ...
    double T1[6], T2[6];
    for (int r=0; r<6; ++r) {
      T1[r] = a*A[i][r]+b*B[i][r];
      T2[r] = d*D[i][k]+e*E[i][k];
    }
    for (int j0=0; j0<3; ++j0)
      for (int k0=0; k0<3; ++k0)
        M[j0+3][k0+3] += f(T1[j0+3], A[i][k0+3], ...);
    for (int j1=0; j1<3; ++j1)
      for (int k1=0; k1<6; ++k1)
        M[j1][k1] += g(T2[k1], A[i][j1+3], ...);
  }
}
```

(b) The code after symbolic execution took place

Fig. 1: Simplified excerpt of local assembly code from a Burgers form using vector-valued basis functions, before and after symbolic execution is performed to rewrite the iteration space

ther in the form of code annotation, if the input to COFFEE were provided as pure C (as shown in Figure 1(a)), or by suitably decorating basis functions nodes, if the input were an abstract syntax tree. Then, the rewrite rules are applied and each statement is executed symbolically. For example, consider the assignment `T2[r] = d*D[i][k]+e*E[i][k]` in Figure 1(a). Array `D` has non-zero-valued columns in the range $NZ_D = [3, 5]$; we also assume array `E` has non-zero-valued columns in the range $NZ_E = [0, 2]$. Multiplications by scalar values do not affect the propagation of non-zero-valued columns. On the other hand, when symbolically executing the sum of the two operands `d*D[i][k]` and `e*E[i][k]`, we track that the target identifier `T2` will have non-zero-valued columns in the range $NZ_E \| NZ_D = [0, 5]$. Eventually, exploiting the NZ information computed and associated with each identifier, we split the original assembly expression into multiple sets of sub-expressions, each set characterized by the same range of non-zero-valued columns. In our example, assuming that $NZ_{T1} = [3, 5]$ and $NZ_A = [3, 5]$, there are two of such sets, which leads to the generation of two distinct iteration spaces (one for each set), as in Figure 1(b).

5.2. 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 1(b). 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.

5.3. Vector-promotion of Integration Quantities

....

6. PERFORMANCE EVALUATION

6.1. 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.

6.2. 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 5.1, 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.

7. 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. 2006. A Compiler for Variational Forms. *ACM Trans. Math. Softw.* 32, 3 (Sept. 2006), 417–444. DOI:http://dx.doi.org/10.1145/1163641.1163644

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

Diego Rossinelli, Babak Hejazialhosseini, Panagiotis Hadjidoukas, Costas Bekas, Alessandro Curioni, Adam Bertsch, Scott Futral, Steffen J. Schmidt, Nikolaus A. Adams, and Petros Koumoutsakos. 2013. 11 PFLOP/s Simulations of Cloud Cavitation Collapse. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 3, 13 pages. DOI:http://dx.doi.org/10.1145/2503210.2504565

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).

Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul D. Hovland. 2010. Speeding Up Nek5000 with Autotuning and Specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, New York, NY, USA, 253–262. DOI:http://dx.doi.org/10.1145/1810085.1810120