A Domain-Specific Optimizing Compiler for Solving Partial Differential Equations

Fabio Luporini
Department of Computing
Imperial College London
London, UK
Email: f.luporini12@imperial.ac.uk

Florian Rathgeber
Department of Computing
Imperial College London
London, UK
Email: f.rathgeber10@imperial.ac.uk

David Ham
Department of Computing
Imperial College London
London, UK
Email: d.ham@imperial.ac.uk

***Abstract*—The abstract goes here.**

I. Introduction

In many fields, like computational fluid dynamics, computational electromagnetics, and structural mechanics, programs model phenomena by means of partial differential equations (PDEs). Numerical techniques, like Finite Volume Method (FVM) and Finite Element Method (FEM), are widely employed in real-world applications to approximate solutions of PDEs. Unstructured meshes are often used to discretize the domain of a PDE, since their geometric flexibility allows solvers to be extrimely effective. However, they are also one of the main responsible for performance degradation, due to the need for indirect accesses (e.g. `A[B[i]]`) to access mesh components. A typical problem, therefore, is that implementing a scientific application requires a great effort, other than domain knowledge, and even more time is needed to optimize their execution time.

An unstructured mesh discretizes the domain of the equation into two- or three-dimensional cells (or "elements"), within which the solution is sought. As the number of elements can be of the order of millions, a major issue is the time required to perform the numerical method, which can be hours or days. To address this issue, domain-specific languages (DSLs), which ease the programming burden and enable highly-efficient parallel execution, have been developed. The successful porting of Hydra, a CFD industrial application devised by Rolls Royce for turbomachinery design (based on FVM, roughly 50000 lines of code and mesh sizes that can be over 100 millions edges), to the OP2 DSL [2], demonstrates the effectiveness of the DSL approach to implementing PDEs solvers [**?**].

OP2, as well as Lizst [1], adopt a kernel-oriented programming model, in which the computation semantics is expressed through self-contained functions (or "kernels"). A kernel is applied to all elements in a set of mesh components (e.g. edges, vertices, elements), with an implicit synchronization between the application of two consecutive kernels. On commodity multi-cores, a kernel is executed sequentially by a thread, while parallelism is achieved partitioning the mesh and assigning each partition to a thread. Together with the indirections problem, kernel optimization is one of the major concerns in unstructured mesh applications. In this paper, we tackle this problem by proposing a domain-driven optimization strategy for a class of kernels used in FEM.

We focus on Finite Element Local Assembly ("assembly", in the following), a fundamental step of FEM that covers an important fraction of the overall computation run-time, typically in the range 30%-60%. During the assembly phase, the solution of the PDE is approximated by executing a suitable kernel over all elements in the discretized domain. A kernel's working set is usually small enough to fit the L1 cache; it might need L2 cache when (very) high-order methods are employed to improve the accuracy of the solution. However, we do not consider the latter case. An assembly kernel is characterized by the presence of an affine, usually non-perfect loop nest, where individual loops are rather small (the trip count rarely exceeds 60, with a minimum value of 3, depending on the order of the method). With such small kernels, our work focuses on aspects like minimization of floating-point operations, register allocation and instruction-level parallelism, especially in the form of SIMD vectorisation.

Our study is conducted in the context of Firedrake, a multilayered system for the automated solution of PDEs through FEM. PDEs are written in Firedrake using a domain-specific language called Unified Form Language (UFL). The high-level UFL notation is used by the Fenics Form Compiler (FFC) to produce assembly kernels. At the core of Firedrake [**?**] there is PyOP2 [2] - a python implementation of the OP2 abstraction - which triggers compilation of kernels using an available vendor compiler, and eventually manages parallel execution over the unstructured mesh. We exploit the domain knowledge and the structure inherent to FFC's assembly kernels to develop an optimizing compiler for all problems that are expressible in Firedrake.

The compiler we have developed is integrated with PyOP2, and already used in the released version of Firedrake [**?**]. This allows us to evaluate the proposed code transformations in a range of real-world problems, employing different low-order methods (from $p = 1$ to $p = 4$; the higher, the more expensive is a kernel, but better is the accuracy), and varying the topology of the mesh (2D, 3D, 3D hybrid structured-unstructured, which in turn affect the kernel cost). Performance results show that compiler-generated code for non-trivial assembly kernels, if not adequately transformed, is sub-optimal. A cost-model-driven sequence of source-to-source code transformations, aimed at improving register allocation and register data locality, can result in performance improvements up to 33% over "softly-optimized" code (i.e. where only basic transformations are performed, such as auto-vectorizable loop-invariant code

```
void helmholtz(double A[3][3], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W3[3] = {...}
  static const double X_D10[3][3] = {{...}}
  static const double X_D01[3][3] = {{...}}

  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      for (int k = 0; k<3; k++)
        A[j][k] += ((Y[i][k]*Y[i][j]+
           +((K[1]*X_D10[i][k]+K[3]*X_D01[i][k])*
           *(K[1]*X_D10[i][j]+K[3]*X_D01[i][j]))+
           +((K[0]*X_D10[i][k]+K[2]*X_D01[i][k])*
           *(K[0]*X_D10[i][j]+K[2]*X_D01[i][j])))*
           *det*W3[i]);
}
```

Algorithm 1: Local assembly code generated by FFC for a Helmholtz problem (2D mesh, Lagrange $p = 1$ elements).

```
void burgers(double A[12][12], double **c, double **w) {
  // K, det = Compute Jacobian (c)

  static const double W5[5] = {...}
  static const double X1_D001[5][12] = {{...}}
  static const double X2_D001[5][12] = {{...}}
  //It follows 11 other basis functions definitions.
  ...

  for (int i = 0; i<5; i++) {
    double F0 = 0.0;
    //It follows 10 other declarations (F1, F2,...)
    ...
    for (int r = 0; r<12; r++) {
      F0 += (w[r][0]*X1_D100[ip][r]);
      //It follows 10 analogous statements (F1, F2, ...)
    }
    ...
    for (int j = 0; j<12; j++)
      for (int k = 0; k<12; k++)
        A[j][k] += (..(K[5]*F9)+(K[8]*F10))*Y1[i][j])+
         +(((K[0]*X1_D100[i][k])+(K[3]*X1_D010[i][k])+
         +(K[6]*X1_D001[i][k]))*Y2[i][j]))*F11)+
         +(..((K[2]*X2_D100[i][k])+...+(K[8]*X2_D001[i][k])*
         *(K[2]*X2_D100[i][j])+...+(K[8]*X2_D001[i][j]))..)+
         + <roughly a hundred of sum/muls go here>)..)*
         *det*W5[i]);
  }
}
```

Algorithm 2: Local assembly code generated by FFC for a Burgers problem (3D mesh, Lagrange $p = 1$ elements).

motion, padding and data alignment), and up to 77% over the original FFC kernels. The contribution of this paper is therefore twofold:

- An optimisation strategy for a class of kernels widely-used in scientific applications, namely Finite Element Local Assembly kernels. Our approach exploits domain knowledge to go beyond the limits of both vendor (e.g. *icc*) and research (e.g. those based on the polyhedral model) compilers.
- Automation of such code optimizations in Firedrake, a framework for the resoloution of PDEs through FEMs. This enables performing an in-depth performance evaluation of the proposed optimization strategy in real-world simulations.

The paper is organized as follows. ...

II. Background

Local assembly consists of evaluating so called element stiffness matrix and vector; in this work, we focus on computation of matrices, which is the costly part of the process. A stiffness matrix can be intuitively thought as an approximated representation of the PDE solution in a specific cell of the discretized domain. Quadrature algorithms (based on numerical integration), which follow on from the mathematical description of a stiffness matrix, are widely-used in scientific codes to implement assembly [**?**], [**?**]. The FFC compiler used by Firedrake and Fenics can generate assembly kernels using optimized quadrature schemes. It is worth noticing, however, that problems and considerations similar to that exposed below can be found also in non-FFC assembly codes.

The complexity of FFC-generated kernels depends on the mathematical problem being solved. In simpler cases, the loop nest is perfect, it has short trip counts (in the range 3-15), and the computation reduces to a summation of a few, small outer-product vector-operations. An example is provided in Figure 1, which shows an assembly kernel for a Helmholtz problem, using Lagrange basis functions on 2D elements with polynomial order $p = 1$. In other scenarios, for instance when solving a non-linear problem like Burgers (see Figure 2), the number of arrays involved in the computation of A can be much larger: in this case, 14 unique arrays are accessed, and the same array can be referenced multiple times within the expression. Also, constants evaluated in outer loops (called F in the code), acting as scaling factors of arrays, may be required; trip counts can be larger (proportionally to the order of the method); arrays may be block-sparse. Despite such a big space of possible assembly implementations, it is still possible to identify a domain-specific structure common to all kernels, which can be turned into effective code transformations and simd-vectorization.

The class of kernels we are considering has, in particular, some peculiarities. 1) The computation of the Jacobian, which is the first step of the assembly, is independent of the loop nest; this is not true in general, since the unstructured mesh might have bended elements that require the Jacobian be re-computed every i iteration. 2) The stiffness matrix A is always the result of an outer-product computation along the j and k dimensions. 3) Memory accesses along the three loop dimensions are always 1-stride .4) The j and k loops are interchangable, whereas to permute i it might be necessary to precompute F values and to introduce temporaries. 5) Basis functions arrays (denoted by X, Y, ...) are constants, and many sub-expressions on the right hand side of the stiffness matrix computation depend on just two loops (either i-j or i-k). In Section III we show how to exploit these observations to define a set of systematic, composable optimizations.

One attractive possibility is to transform the loop nest into a sequence of calls to BLAS routines. There are several issues that make achieving this difficult and, potentially, not effective. The first problem concerns automating identification and extraction of matrix-matrix multiplications from the code, which requires a non-trivial analysis of the memory access pattern. Related to this problem is the fact that, in general, many sub-expressions should be pre-computed (i.e. computed once before the loop nest), which leads to using temporaries and, possibly, to working sets that do not fit the L1 cache (for $p > 2$). In addition, using BLAS routines may impact data locality, since the same matrix can appear in multiple products, implying, at least, a certain degree of register spilling. Finally, it is known that BLAS routines perform far from peak performance when the dimension of the matrices is rather small [**?**]. Hand-made implementations of the simple Helmholtz problem ($p = 1, 2, 3, 4$) as a sequence of BLAS ($dgemm$) routines have demonstrated that performance is worse than that shown in Section V. A comprehensive study concerning BLAS optimization is therefore left as further work.

III. Domain-Driven Code Transformations

From inspection of the codes in Figures 1 and 2, it can be noticed that the computation of A involves evaluating many sub-expressions that depend on two iteration variables only. Since symbols in most of these sub-expressions are read-only variables, there is ample space for loop-invariant code motion ($licm$). Vendor compilers apply $licm$, although not in the systematic way we need in our assembly kernels. We want to overcome two deficiencies that both icc and gcc have. First, these compilers can identify sub-expressions that are invariant with respect to the innermost loop only. This is an issue for sub-expressions depending on i-k only, which are not automatically lifted. Second, the hoisted code is scalar, i.e. it is not subjected to auto-vectorization. We work around these limitations by doing source-level $licm$. In addition, we pre-compute all values that an invariant sub-expression assumes along the fastest varying dimension. This is implemented by introducing temporary arrays and new loops in the nest. At the price of both extra memory for storing temporaries and reduced register locality, the gain is that lifted terms can be auto-vectorized, because part of an inner loop. Note that given the short trip counts of our loops, it is important to achieve auto-vectorization of hoisted terms in order to minimize the percentage of scalar instructions, which can be otherwise significant.

Auto-vectorization of assembly code that computes A can be less effective if data are not aligned and if the length of the innermost loop is small compared to the hardware vector length (vl). Data alignment is enforced in two steps. Initially, both arrays and matrices are allocated to addresses that are multiples of vl. Then, matrices are padded by rounding the number of columns to the nearest multiple of vl. For example, assume the original size of a matrix is 3×3, and the underlying machine possesses AVX, which implies $vl = 4$ since a vector register is 256 bits long and our kernels use double-precision floating-point values (64 bits). Then, the padded matrix on this architecture is 3×4. Padding of all matrices involved in the evaluation of the stiffness matrix allows us to safely round the loop trip count to the nearest multiple of vl. This avoids the introduction of a remainder (scalar) loop from the compiler, which would be responsible for inefficient vectorization.

One notable problem assembly kernels are exposed to concerns register allocation and register locality. The critical situation occurs when loop trip counts and number of accessed matrices along the j and k dimensions are such that available vector registers are under-utilized. Due to a kernel's working set fitting the L1 cache, it is remarkably important to optimize register management in order to maximize performance. Loop interchange, unroll, unroll-and-jam, and register tiling, are

```
The PyOP2 Compiler
Input: ast, wrapper, isa
Output: code
// Analyze ast and build optimization plan
it_space = analyze(ast)
if not it_space then
    ast.apply_inter_kernel_vectorization(wrapper)
    return ast.from_ast_to_c()
endif
plan = cost_model(it_space.n_inner_arrays, isa.n_regs)

// Optimize ast based on plan
ast.apply_licm()
if plan.sz_split then
    ast.split(plan.sz_split)
endif
if plan.uaj_factor then
    ast.vr_tile(plan.uaj_factor)
endif
ast.padding()
ast.data_align()
return ast.from_ast_to_c()
```

Algorithm 3: The PyOP2 compiler.

```
Algorithm: ApplyCostModel
Input: n_inner_arrays, n_regs
Output: uaj_factor, sz_split
sz_split = 0
// Compute spltting factor
while n_inner_arrays > n_regs do
    n_inner_arrays = n_inner_arrays / 2
    sz_split = sz_split + 1
endw
n_inner_regs = n_regs / 2
if n_inner_arrays > n_inner_regs or sz_split > 0 then
    // Rely on autovectorization, as there might be not
    enough registers to improve performance by tiling
    return <sz_split, 0>
endif
// Compute unroll-and-jam factor for vector-register
tiling
n_regs_avail = n_regs - n_inner_arrays
uaj_factor = ⌈n_reg_avail / n_inner_arrays⌉
return <sz_split, uaj_factor>
```

Algorithm 4: Procedure to estimate the most suitable unroll-and-jam factor and/or split size.

IV. The PyOP2 Compiler

The compiler structure

The compiler cost model. Used to find out suitable unroll-and-jam factor and split size.

V. Performance Evaluation

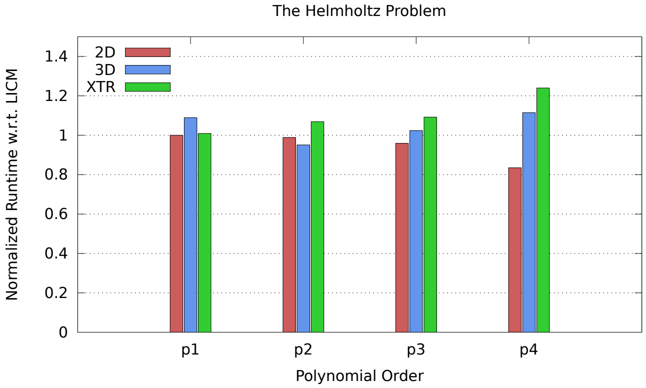

Fig. 1. Helmholtz

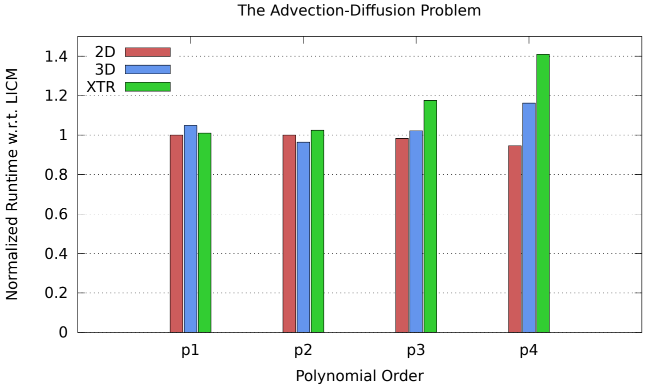

Fig. 2. Advection-Diffusion

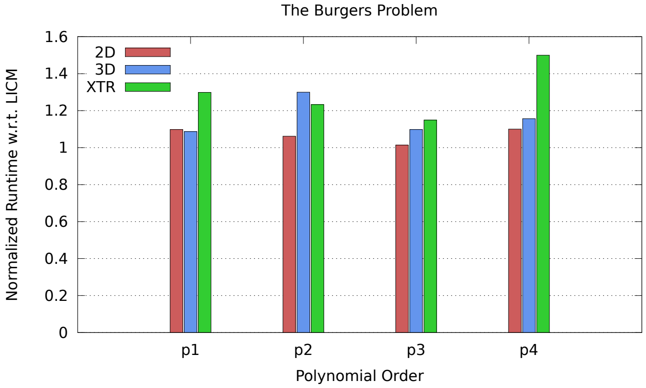

Fig. 3. Burgers

VI. Related Work

VII. Conclusions

Acknowledgment

The authors would like to thank...

References

[1] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[2] G. R. Markall, F. Rathgeber, L. Mitchell, N. Loriant, C. Bertolli, D. A. Ham, and P. H. J. Kelly. Performance portable finite element assembly using PyOP2 and FEniCS. In *Proceedings of the International Supercomputing Conference (ISC) '13*, volume 7905 of *Lecture Notes in Computer Science*, June 2013. In press.