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

Paul H. J. Kelly
Department of Computing
Imperial College London
London, UK
Email: p.kelly@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 introduction of 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 the major concern 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 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 100, 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 developed is integrated with PyOP2, and already used in the release 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 a kernel is, 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

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

III. A DOMAIN-DRIVEN APPROACH TO CODE OPTIMIZATION

IV. THE PYOP2 COMPILER

The compiler structure

1 The PyOP2 Compiler
  Input: ast, wrapper, isa
  Output: code
2 // Analyze ast and build optimization plan
3 it_space = analyze(ast)
4 if not *it_space* **then**
5   ast.apply_inter_kernel_vectorization(wrapper)
      return *ast.from_ast_to_c()*
6 endif
7 plan = cost_model(it_space.n_inner_arrays, isa.n_regs)
8
9 // Optimize ast based on plan
10 ast.apply_licm()
11 if *plan.sz_split* **then**
12   ast.split(plan.sz_split)
13 endif
14 if *plan.uaj_factor* **then**
15   ast.vr_tile(plan.uaj_factor)
16 endif
17 ast.padding()
18 ast.data_align()
19 return *ast.from_ast_to_c()*
        Algorithm 1: The PyOP2 compiler.

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

1 Algorithm: ApplyCostModel
  Input: n_inner_arrays, n_regs
  Output: uaj_factor, sz_split
2 sz_split = 0
3 // Compute spltting factor
4 while *n_inner_arrays > n_regs* **do**
5   n_inner_arrays = n_inner_arrays / 2
6   sz_split = sz_split + 1
7 endw
8 n_inner_regs = n_regs / 2
9 if *n_inner_arrays > n_inner_regs* **or** *sz_split > 0* **then**
10   // Rely on autovectorization, as there might be not
      enough registers to improve performance by tiling
11   **return** *<sz_split, 0>*
12 endif
13 // Compute unroll-and-jam factor for vector-register
  tiling
14 n_regs_avail = n_regs - n_inner_arrays
15 uaj_factor = $\lceil$n_reg_avail / n_inner_arrays$\rceil$
16 return *<sz_split, uaj_factor>*
 Algorithm 2: Procedure to estimate the most suitable unroll-and-jam factor and/or split size.

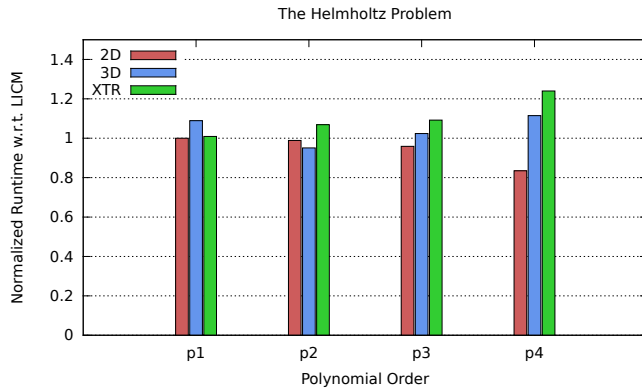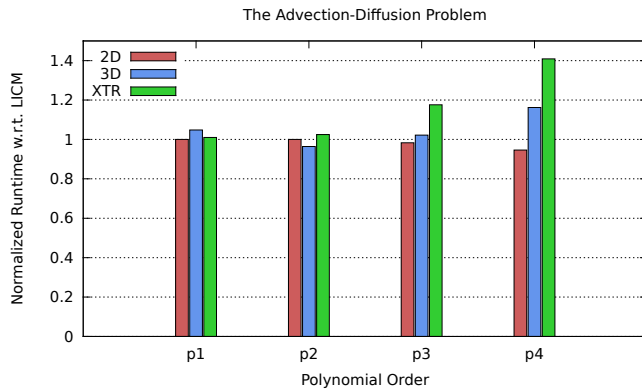## V. Performance Evaluation

Fig. 1. Helmholtz
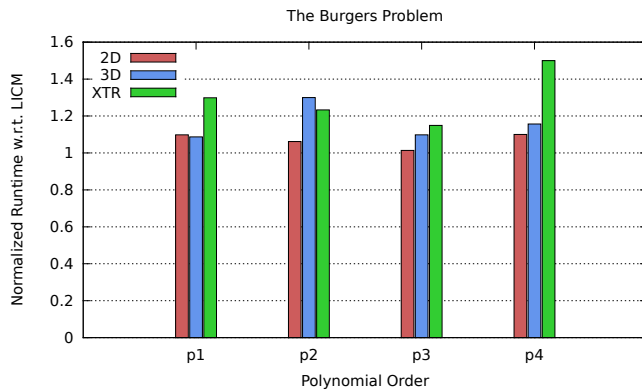

Fig. 2. Advection-Diffusion

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.