Domain-Driven Code Transformation 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 are required that model phenomena. A feature common to these scientific computations is that they involve the resolution 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 PDE domain, since their geometric flexibility allows such PDE solvers to be extrimely effective. However, they are also 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 represents a discretization of the equation domain into two- or three-dimensional cells (or “elements”), which define the smallest area 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 of the order of hours or days. OP2 [2] and Lizst [1] are Domain Specific Languages that target mesh-based computations by allowing programmers to achieve highly-efficient parallel execution. 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 a mesh size that can be over 100 millions edges), to OP2, demonstrates the effectiveness of this framework for implementing PDEs solvers [**?**]. Both OP2 and Lizst adopt a kernel-oriented programming model: the computation semantics is expressed through self-contained functions (or “kernels”), each function applied in parallel to all items in a specific set of mesh components (e.g. edges, vertices, elements), with an implicit synchronization between the application of two different functions. Together with the indirections problem, having systematic optimizations for the kernels is the major concern in unstructured mesh applications.

In this paper, we describe a model-driven optimization strategy for a class of kernels used in Finite Element computations. The working set of these kernels is usually small enough to fit the L1 cache, and definitely fits the L2 cache when high-order methods are employed to improve the accuracy of the solution. However, the latter case is not considered in our work. This implies that main actors of our study will be register allocation and instruction-level parallelism, especially in the form of SIMD vectorisation.

II. Background

III. A Domain-Driven Approach to Code Optimization

IV. The PyOP2 Compiler

The compiler structure

```
**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 1: The PyOP2 compiler.

The compiler cost model. Used to find out tile size and split size.

```
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 = $\lceil$n_reg_avail / n_inner_arrays$\rceil$
return <sz_split, uaj_factor>
```

Algorithm 2: Procedure to estimate the most suitable unroll-and-jam factor and/or split size.

V. Performance Evaluation

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.