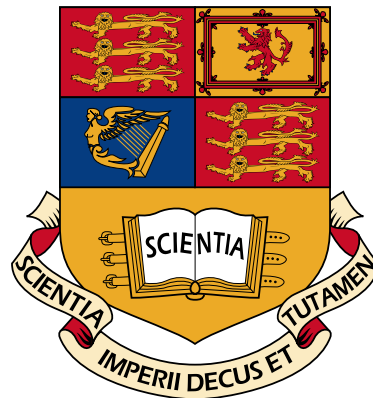Imperial College London

Department of Computing

Bho

Fabio Luporini

September 2014

Supervised by: Dr. David A. Ham, Prof. Paul H. J. Kelly

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing
of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Fabio Luporini

Abstract

TODO

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

In many fields, such as computational fluid dynamics, computational electromagnetics and structural mechanics, phenomena are modelled by partial differential equations (PDEs). Unstructured meshes, which allow an accurate representation of complex geometries, are often used to discretize their computational domain. Numerical techniques, like the finite volume method and the finite element method, approximate the solution of a PDE by applying suitable numerical operations, or kernels, to the various entities of the unstructured mesh, such as edges, vertices, or cells. On standard clusters of multicores, typically, a kernel is executed sequentially by a thread, while parallelism is achieved by partitioning the mesh and assigning each partition to a different node or thread. Such an execution model, with minor variations, is adopted, for instance, in Markall et al. [2013], Logg et al. [2012], AMCG [2010], DeVito et al. [2011], which are examples of frameworks specifically thought for writing numerical methods for PDEs.

The time required to execute these unstructured-mesh-based applications is a fundamental issue. An equation domain needs to be discretized into an extremely large number of cells to obtain a satisfactory approximation of the solution, possibly of the order of trillions (e.g. Rossinelli et al. [2013]), so applying numerical kernels all over the mesh is expensive. For example, it is well-established that mesh resolution is crucial in the accuracy of numerical weather forecasts; however, operational centers have a strict time limit in which to produce a forecast - 60 minutes in the case of the UK

Met Office - so, executing computation- and memory-efficient kernels has a direct scientific payoff in higher resolution, and therefore more accurate predictions. Motivated by this and analogous scenarios, this thesis studies, formalizes, and implements a number of code transformations to improve the performance of real-world scientific applications using numerical methods over unstructured meshes.

1.2 Contributions

Besides developing novel compilers theory, contributions of this thesis come in the form of tools that have been, and currently are, used to accelerate scientific computations, namely:

- **A run-time library, which can be regarded as a prototype compiler, capable of optimizing sequences of non-affine loops by fusion and automatic parallelization. The library has been used to speed up a real application for tsunami simulations as well as other representative benchmarks.**

  One limiting factor to the performance of unstructured mesh applications is imposed by the need for indirect memory accesses (e.g. `A[B[i]]`) to read and write data associated with the various entities of the discretized domain. For example, when executing a kernel that numerically computes an integral over a cell, which is common in a finite element method, it may be necessary to read the coordinates of the adjacent vertices; this is achieved by using a suitable indirection array, often referred to as "map", that connects cells to vertices. The problem with indirect memory accesses is that they break several hardware and compiler optimizations, including prefetching and standard loop blocking (or tiling). A first contribution of this thesis is the formalization and development of a technique, called "generalized sparse tiling", that aims at increasing data locality by fusing loops in which indirections are used. The challenges are 1) to determine if and how a sequence of loops can be fused, and 2) doing it efficiently, since the fusability analysis must be performed at run-time, once the maps are available (i.e. once the mesh topology has been read from disk).

2

- **A fully-operating compiler, named COFFEE[1], which optimizes numerical kernels solving partial differential equations using the finite element method. The compiler is integrated with the Firedrake system (Firedrake contributors [2014]).**

The second contribution, in the context of the finite element method, is about optimizing the computation of so called local element matrices, which can be responsible for a significant fraction of the overall computation run-time. This is a well-known problem, and several studies can be found in the literature, among which Russell and Kelly [2013], Olgaard and Wells [2010], Knepley and Terrel [2013], Kirby et al. [2005]. With respect to these studies, we propose a novel set of composable, model-aware code transformations explicitly targeting, for the first time, instruction-level parallelism - with emphasis on SIMD vectorization - and register locality. These transformations are automated in COFFEE.

1.3 Thesis Statement

1.4 Motivation

1.5 Contributions

1.6 Statement of Originality

1.7 Dissemination

1.8 Thesis Outline

---

[1]COFFEE stands for COmpiler For FinitE Element local assembly.

Chapter 2

Background

2.1 bho

Chapter 3

Sparse Tiling

3.1 bho

Chapter 4

Cross-loop Optimization of Arithmetic Intensity for Finite Element Local Assembly

4.1 Introduction and Motivations

In many fields such as computational fluid dynamics, computational electromagnetics and structural mechanics, phenomena are modelled by partial differential equations (PDEs). Numerical techniques, like the finite volume method and the finite element method, are widely employed to approximate solutions of these PDEs. Unstructured meshes are often used to discretize the computational domain, since they allow an accurate representation of complex geometries. The solution is sought by applying suitable numerical operations, or kernels, to the entities of a mesh, such as edges, vertices, or cells. On standard clusters of multicores, typically, a kernel is executed sequentially by a thread, while parallelism is achieved by partitioning the mesh and assigning each partition to a different node or thread. Such an execution model, with minor variations, is adopted, for example, in Markall et al. [2013], Logg et al. [2012], AMCG [2010], DeVito et al. [2011].

The time required to apply the numerical kernels is a major issue, since the equation domain needs to be discretized into an extremely large number

of cells to obtain a satisfactory approximation of the PDE, possibly of the order of trillions, as in Rossinelli et al. [2013]. For example, it has been well established that mesh resolution is critical in the accuracy of numerical weather forecasts. However, operational forecast centers have a strict time limit in which to produce a forecast - 60 minutes in the case of the UK Met Office. Producing efficient kernels has a direct scientific payoff in higher resolution, and therefore more accurate, forecasts. Computational cost is a dominant problem in computational science simulations, especially for those based on finite elements, which are the subject of this work. In this chapter, we address, in particular, the well-known problem of optimizing the local assembly phase of the finite element method, which can be responsible for a significant fraction of the overall computation run-time, often in the range 30-60% ??Knepley and Terrel [2013], ?.

During the assembly phase, the solution of the PDE is approximated by executing a problem-specific kernel over all cells, or elements, in the discretized domain. We restrict our focus to relatively low order finite element methods, in which an assembly kernel's working set is usually small enough to fit the L1 cache. Low order methods are by no means exotic: they are employed in a wide variety of fields, including climate and ocean modeling, computational fluid dynamics, and structural mechanics. The efficient assembly of high order methods such as the spectral element method ? requires a significantly different loop nest structure. High order methods are therefore excluded from our study.

An assembly kernel is characterized by the presence of an affine, often non-perfect loop nest, in which individual loops are rather small: their trip count rarely exceeds 30, and may be as low as 3 for low order methods. In the innermost loop, a problem-specific, compute intensive expression evaluates a two dimensional array, representing the result of local assembly in an element of the discretized domain. With such a kernel structure, we focus on aspects like the minimization of floating-point operations, register allocation and instruction-level parallelism, especially in the form of SIMD vectorization.

We aim to maximize our impact on the platforms that are realistically used for finite element applications, so we target conventional CPU architectures rather than GPUs. The key limiting factor to the execution on GPUs is the stringent memory requirements. Only relatively small problems fit in

a GPU memory, and support for distributed GPU execution in general purpose finite element frameworks is minimal. There has been some research on adapting local assembly to GPUs (mentioned later), although it differs from ours in several ways, including: (i) not relying on automated code generation from a domain-specific language (explained next), (ii) testing only very low order methods, (iii) not optimizing for cross-loop arithmetic intensity (the goal is rather effective multi-thread parallelization). In addition, our code transformations would drastically impact the GPU parallelization strategy, for example by increasing a thread's working set. For all these reasons, a study on extending the research to GPU architectures is beyond the scope of this work. In Section 4.10, however, we provide some intuitions about this research direction.

Achieving high-performance on CPUs is non-trivial. The complexity of the mathematical expressions, often characterized by a large number of operations on constants and small matrices, makes it hard to determine a single or specific sequence of transformations that is successfully applicable to all problems. Loop trip counts are typically small and can vary significantly, which further exacerbates the issue. We will show that traditional vendor compilers, such as *GNU's* and *Intel's*, fail at exploiting the structure inherent such assembly expressions. Polyhedral-model-based source-to-source compilers, for instance **?**, can apply aggressive loop optimizations, such as tiling, but these are not particularly helpful in our context, as explained next.

We focus on optimizing the performance of local assembly operations produced by automated code generation. This technique has been proved successful in the context of the FEniCS Logg et al. [2012] and Firedrake Firedrake contributors [2014] projects, become incredibly popular over the last years. In these frameworks, a mathematical model is expressed at high-level by means of a domain-specific language and a domain-specific compiler is used to produce a representation of local assembly operations (e.g. C code). *Our aim is to obtain close-to-peak performance in all of the local assembly operations that such frameworks can produce.* Since the domain-specific language exposed to the users provide as constructs generic differential operators, an incredibly vast set of PDEs, possibly arising in completely different domains, can be expressed and solved. A compiler-based approach is, therefore, the only reasonable option to the problem of optimizing local assembly

11

operations.

Several studies have already tackled local assembly optimization in the context of automated code generation. In **?**, it is shown how this technique can be leveraged to introduce domain-specific optimizations, which a user cannot be expected to write "by hand". **?** and **?** have studied, instead, different optimization techniques based on a mathematical reformulation of the local assembly operations. The same problem has been addressed recently also for GPU architectures, for instance in Knepley and Terrel [2013], **?**, and **?**. With our study, we make clear step forward by showing that different PDEs, 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. As discussed in the following sections, our optimization strategy is quite different from those in previous work, although we reuse and leverage some of the ideas formulated in the available literature.

We present a novel structured approach to the optimization of automatically-generated local assembly kernels. We argue that for complex, realistic PDEs, peak performance can be achieved only by passing through a two-step optimization procedure: 1) expression rewriting, to minimize floating point operations, 2) and code specialization, to ensure effective register utilization and instruction-level parallelism, especially SIMD vectorization.

Expression rewriting consists of a framework capable of minimizing arithmetic intensity and optimize for register pressure. Our contribution is twofold:

- *Rewrite rules for assembly expressions.* The goal is to the reduce the computational intensity of local assembly kernels by rescheduling arithmetic operations based on a set of rewrite rules. These aggressively exploit associativity, distributivity, and commutativity of operators to expose loop-invariant sub-expressions and SIMD vectorization opportunities to the code specialization stage. While rewriting an assembly expression, domain knowledge is used in several ways, for example to avoid redundant computation.

- *An algorithm to deschedule useless operations.* Relying on symbolic execution, this algorithm restructures the code so as to skip useless

12

arithmetic operations, for example multiplication by scalar quantities which are statically known to be zero. One problem is to transform the code while preserving code vectorizability, which is solved by resorting to domain-knowledge.

Code specialization's goal is to apply transformations to maximize the exploitation of the underlying platform's resources, e.g. SIMD lanes. We provide a number of contributions:

- *Padding and data alignment.* The small size of the loop nest (integration, test, and trial functions loops) require all of the involved arrays to be padded to a multiple of the vector register length so as to maximize the effectiveness of SIMD code. Data alignment can be enforced as a consequence of padding.

- *Vector-register Tiling.* Blocking at the level of vector registers, which we perform exploiting the specific memory access pattern of the assembly expressions (i.e. a domain-aware transformation), improves data locality beyond traditional unroll-and-jam optimizations. This is especially true for relatively high polynomial order (i.e. greater than 2) or when pre-multiplying functions are present.

- *Expression Splitting.* In certain assembly expressions the register pressure is significantly high: when the number of basis functions arrays (or, equivalently, temporaries introduced by loop-invariant code motion) and constants is large, spilling to L1 cache is a consequence for architectures with a relatively low number of logical registers (e.g. 16/32). We exploit sum's associativity to "split" the assembly expression into multiple sub-expressions, which are computed individually.

- *An algorithm to generate calls to BLAS routines.*

- *Autotuning.* We implement a model-driven, dynamic autotuner that transparently evaluates multiple sets of code transformations to determine the best optimization strategy for a given PDE. The main challenge here is to build, for a generic problem, a reasonably small search space that comprises most of the effective code variants.

Expression rewriting and code specialization have been implemented in a

compiler, COFFEE[1], fully integrated with the Firedrake framework **?**. Besides separating the mathematical domain, captured by a domain-specific compiler at an higher level of abstraction, from the optimization process, COFFEE also aims to be platform-agnostic. The code transformations occur on an intermediate representation of the assembly operation, which is ultimately translated into platform-specific code. Domain knowledge is exploited in two ways: for simplifying the implementation of code transformations and to make them extremely effective. Domain knowledge is conveyed to COFFEE from the higher level through suitable annotations attached to the input. For example, when the input is in the form of an abstract syntax tree produced by the higher layer, specific nodes are decorated so as to drive the optimization process. Although COFFEE has been thought of as a multi-platform optimizing compiler, our performance evaluation so far has been restricted to standard CPU platforms only. We emphasize once more, however, that all of the transformations applicable would work on generic accelerators as well.

To demonstrate the effectiveness of our approach, we provide an extensive and unprecedented performance evaluation across a number of real-world PDEs 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 so called pre-multiplying functions. To clearly distinguish the improvements achieved by COFFEE, we will compare, for each examined PDE, four sets of code variants: 1) unoptimized code, i.e. a local assembly routine as returned from the domain-specific compiler; 2) code optimized by FEniCS, i.e. the work in **?**; 3) code optimized by expression rewriting and code specialization as described in this paper. Notable performance improvements of 3) over 1) and 2) are reported and discussed.

---

[1]COFFEE stands for COmpiler For Finit Element local assEmbly.

14

4.2 Preliminaries

4.2.1 Overview of the Finite Element Method

4.2.2 From Math to Actual Code

4.3 On the Space of Possible Code Optimizations

4.3.1 Outline of COFFEE

4.4 Expression Rewriting

4.5 Code Specialization

4.6 Composing Optimizations

4.7 Design and Implementation of COFFEE

4.7.1 Input ad Output: the Integration with Firedrake

4.7.2 Structure

4.7.3 Conveying Domain-Specific Knowledge

4.8 Performance Analysis

4.8.1 Contribution of Basic Optimizations

4.8.2 Evaluation in Forms of Increasing Complexity

4.8.3 Full Application Study

4.9 Related Word

4.10 Generality of the Approach and Applicability to Other Domains

4.11 Conclusion

Chapter 5

Conclusion

5.1 bho

Bibliography

AMCG. *Fluidity Manual.* Applied Modelling and Computation Group, Department of Earth Science and Engineering, South Kensington Campus, Imperial College London, London, SW7 2AZ, UK, version 4.0-release edition, November 2010. available at `http://hdl.handle.net/10044/1/7086` .

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. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063396. URL `http://doi.acm.org/10.1145/2063384.2063396`.

Firedrake contributors. The Firedrake Project. `http://www.firedrakeproject.org`, 2014.

Robert C. Kirby, Matthew Knepley, Anders Logg, and L. Ridgway Scott. Optimizing the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 27(3):741–758, October 2005. ISSN 1064-8275. doi: 10.1137/040607824. URL `http://dx.doi.org/10.1137/040607824`.

Matthew G. Knepley and Andy R. Terrel. Finite element integration on gpus. *ACM Trans. Math. Softw.*, 39(2):10:1–10:13, Febru-

ary 2013. ISSN 0098-3500. doi: 10.1145/2427023.2427027. URL
http://doi.acm.org/10.1145/2427023.2427027.

Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated So-
lution of Differential Equations by the Finite Element Method*. Springer,
2012. ISBN 978-3-642-23098-1. doi: 10.1007/978-3-642-23099-8.

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.

Kristian B. Olgaard and Garth N. Wells. Optimizations for quadra-
ture representations of finite element tensors through automated
code generation. *ACM Trans. Math. Softw.*, 37(1):8:1–8:23, Jan-
uary 2010. ISSN 0098-3500. doi: 10.1145/1644001.1644009. URL
http://doi.acm.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. 11 pflop/s
simulations of cloud cavitation collapse. In *Proceedings of the Interna-
tional Conference on High Performance Computing, Networking, Stor-
age and Analysis*, SC '13, pages 3:1–3:13, New York, NY, USA, 2013.
ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2504565. URL
http://doi.acm.org/10.1145/2503210.2504565.

Francis P. Russell and Paul H. J. Kelly. Optimized code generation for finite
element local assembly using symbolic manipulation. *ACM Transactions
on Mathematical Software*, 39(4), 2013.

20