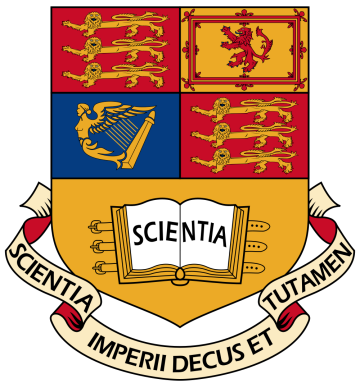
Imperial College London
Department of Computing




# Exploring Optimisations for the Local Assembly phase of Finite Element Methods on GPUs

Hector Dearman

June 2015

Supervised by Prof. Paul H. J. Kelly and Fabio Luporini

# Abstract

Finite Element Methods (FEM) are ubiquitous in science and engineering where they are used in fields as diverse as structural analysis, ocean modeling and bioengineering. FEM allow us to find approximate solutions to a system of partial differential equations over an unstructured mesh. The first phase of solving a FEM problem, local assembly, involves computing a tensor for every element in the mesh. Local assembly is extremely data-parallel, each entry in each tensor may be computed independently, making local assembly an excellent target for General Purpose Graphics Processing Units.

We systematically investigate optimisations to improve the performance of the local assembly phase of FEM on GPUs for a broad range of problems. We look at four classes of optimisations: effective use of constant memory, tuning the kernel launch parameters, using multiple threads per element and loop unrolling.

The optimisations are implemented in the Firedrake toolchain, particularly in PyOP2 and COFFEE, and the performance improvement of each optimisation is measured using three representative benchmarks. In order to ensure our results are robust we consider each of these benchmarks in the context of a variety of element shapes and polynomial degrees of the basis functions. Combining these optimisations, we achieve speed increases of up to 35 times compared to Firedrake's current performance on some benchmarks and an average increase of 13 times across all benchmarks. Finally, we measure the absolute performance of the combined optimisations, showing that we achieve up to 78% of peak FLOPs on some benchmarks and an average of 57% of peak FLOPs across all benchmarks on an NVIDA GRID K520.

## Acknowledgements

I would like to thank Prof. Paul Kelly and Fabio Luporini, who were unfailingly generous with their time and knowledge and who helped me navigate the deep waters of Finite Element Methods, making this project possible. Thanks go also to my family, who supported me always, to my friends, who survived Imperial with me and to Sam, who did both.

# Contents


**1 Introduction** **10**

1.1 Objectives . . . 11

1.2 Contributions . . . 12

**2 Background** **14**

2.1 Finite Element Method . . . 14

2.1.1 Local Matrix Approach vs. Addto Algorithm . . . 15

2.1.2 Local Assembly Structure . . . 15

2.1.3 Meshes . . . 16

2.1.4 Basis Functions . . . 16

2.1.5 Jacobian . . . 16

2.2 Firedrake Toolchain . . . 17

2.2.1 UFL . . . 17

2.2.2 FFC . . . 18

2.2.3 PyOP2 . . . 18

2.2.4 PETSc . . . 18

2.3 COFFEE . . . 18

2.4 GPU Architecture . . . 18

2.5 GPU Programming Models . . . 19

2.5.1 CUDA . . . 19

2.5.2 OpenCL . . . 21

**3 Related Work** **23**

**4 Experimental Methodology** **25**

4.1 Timing Benchmarks . . . 25

4.2 Profiling Benchmarks . . . 26

4.3 Selection and range of benchmarks . . . 26

4.3.1 Mass . . . 27

4.3.2 Helmholtz . . . 27

4.3.3 Elasticity . . . 27

4.4 Benchmark Parameters . . . 28

4.4.1 Polynomial Degree of Basis Functions . . . 28

4.4.2 Mesh type . . . 28

4.4.3 Mesh size . . . 29
4.4.4 Overview . . . 30
4.5 Correctness . . . 30
4.6 Hardware . . . 33

**5 Choice of Optimisations** **34**

**6 Investigation** **35**
6.1 Constant Hoisting . . . 35
6.1.1 Current Status . . . 36
6.1.2 Hypothesis . . . 36
6.1.3 Experiment . . . 36
6.1.4 Implementation . . . 36
6.1.5 Discussion . . . 39
6.2 Parameter Tuning . . . 42
6.2.1 Current Status . . . 44
6.2.2 Hypothesis . . . 45
6.2.3 Experiment . . . 45
6.2.4 Implementation . . . 45
6.2.5 Discussion . . . 46
6.3 Loop Unrolling . . . 46
6.3.1 Current Status . . . 48
6.3.2 Hypothesis . . . 48
6.3.3 Experiment . . . 48
6.3.4 Discussion . . . 48
6.4 Multiple Threads per Element . . . 49
6.4.1 Number of threads per element . . . 53
6.4.2 Special Case Loop Flattening . . . 59
6.4.3 Chunked vs. Coalesced . . . 60
6.4.4 Parameter Tuning and Multiple Threads per Element . . . 63

**7 Conclusion** **67**
7.1 Contributions . . . 68
7.2 Conclusion . . . 69
7.3 Future Work . . . 69

**Glossary** **72**

# List of Tables


2.1 CUDA vs. OpenCL Keywords [5] . . . . . 21
2.2 CUDA vs. OpenCL Terminology [5] . . . . . 22

4.1 Benchmarks . . . . . 31
4.2 GPU Characteristics . . . . . 33

6.1 Memory Utilisation in $\{\text{mass}, \text{quadrilateral}, \text{degree } 4\}$ on a 10,000 element mesh . . . . . 41
6.2 Benchmarks with small performance drops after constant hoisting . . . . . 42
6.3 Benchmarks with large performance drops after constant hoisting . . . . . 42
6.4 Parameter space . . . . . 45
6.5 Use of `pragma unroll`[22] . . . . . 48
6.6 Summary of `pragma unroll` experiment . . . . . 49
6.7 Speedups for Special Case Loop Flattening vs. Loop Flattening . . . . . 61
6.8 Memory Access Efficiency for $\{\text{mass,triangle,4, o, degree } n\}$ a 10,000 element mesh . . . . . 62

7.1 Best combined optimisation for each benchmark . . . . . 71

# List of Figures


2.1 A 2D unstructured mesh with triangular elements . . . . . 14

4.1 Times recored by `cuda_kernel` timer vs. those recored by `parloop_kernel` for every experiment conducted for this report. $y = 1.00027221902x + 0.000236040794433$ . . . . . 26
4.2 Wrieframes of diffrent mesh types . . . . . 29
4.3 'Golden' matrix `mass-unittriangle-10-1.npy` . . . . . 32

6.1 Constant hoisting speedup (compared to Firedrake's current implementation) for each benchmark . . . . . 39
6.2 Constant Hoisting speedup vs. Basis Function Size . . . . . 40
6.3 Screen shot of CUDAs Occupancy Calculator . . . . . 44
6.4 The average speedup (compared to the post-Constant Hoisting baseline) across all benchmarks for each parameter set. . . . . 47
6.5 Maximum speedup compared to the post-Constant Hoisting baseline for each benchmark after a parameter sweep. . . . . 47
6.6 Speedup per benchmark for the parameters $\{\text{blocksize} = 128, \text{partition size} = \frac{1}{2}, \text{blocks per SM} = 4\}$ . . . . . 47
6.7 Speedup of unrolling no loops, unrolling only $ip$ loop, unrolling $ip$ and $k$ loops and unrolling all three loops with pragma unroll, compared to post-Constant Hoisting baseline for each benchmark. . . . . 50
6.8 Current strategy for parallelisation . . . . . 51
6.9 Chunked strategy for parallelisation . . . . . 51
6.10 Coalesced chunked strategy for parallelisation . . . . . 52
6.11 Allocating five threads to a nine entry tensor under the 'coalesced' and 'chunked' strategies . . . . . 56
6.12 Speedup for each benchmark under the 'coalesced' scheme (compared to post-Constant Hoisting performance) for 1 to 128 threads . . . . . 58
6.13 Speedup for each benchmark under the 'chunked' scheme (compared to post-Constant Hoisting performance) for 1 to 128 threads . . . . . 58
6.14 Theoretical speedup for each benchmark for 1 to 128 threads . . . . . 58
6.15 Illustration of the how the chunked (top) and coalesced (bottom) schemes access a six entry local matrix with three threads. . . . . 62

6.16 The speedup for the ‘coalesced gcd10’ scheme compared to the ‘chunked gcd10’ scheme across all benchmarks. . . . . 64
6.17 The speedup for the ‘coalesced gcd10’ scheme compared to the post-constant hoisting baseline. . . . . 64
6.18 The speedup for the ‘chunked gcd10’ scheme compared to the scheme compared to the post-constant hoisting baseline. . . . . 64
6.19 The average speedup of the ‘coalesced gcd10’ scheme (compared to the post-Constant Hoisting baseline) across all benchmarks for each parameter set . . . . . 66
6.20 Maximum speedup of the ‘coalesced gcd10’ scheme compared to the post-constant hoisting baseline for each benchmark after a parameter sweep . . . 66
6.21 Maximum speedup of the ‘coalesced gcd10’ scheme after a parameter sweep compared to the best post-constant hoisting baseline after a parameter sweep 66

7.1 Best achieved speedup compared to Firedrake’s current performance for each benchmark . . . . . 67
7.2 FLOPs for best combination of the optimisations for each benchmark . . . 67

# Listings


2.1 Loop nest structure of a local assembly kernel . . . 15
2.2 Example of UFL . . . 17
2.3 CUDA Kernel for adding vectors . . . 19
2.4 OpenCL Kernel for adding vectors . . . 21

4.1 Mass Benchmark . . . 27
4.2 Helmholtz Benchmark . . . 27
4.3 Elasticity Benchmark . . . 28

6.1 Example of `__constant__` qualifier . . . 35
6.2 Example of `__constant__` qualifier with literal data . . . 36
6.3 Mass benchmark kernel . . . 37
6.4 Mass benchmark kernel after constant hoisting . . . 38
6.5 {mass, quadrilateral, degree 4} assembly snippet . . . 41
6.6 {mass, quadrilateral, degree 4} assembly snippet after Constant Hoisting . . . 41
6.7 Example of launch bounds in CUDA C . . . 44
6.8 Loop nest structure of a local assembly kernel . . . 49
6.9 Example loop nest . . . 53
6.10 Collapsed loop nest . . . 53
6.11 Chunked Strategy Example . . . 53
6.12 Coalesced Stratagy Example . . . 53
6.13 Critical Section of kernel wrapper for benchmark {mass, quadrilateral, degree 4} 54
6.14 Kernel Wrapper Chunked Stratagy . . . 55
6.15 Kernel Wrapper Coalesced Stratagy . . . 55
6.16 With redundant loop . . . 59
6.17 Without redundant loop . . . 59
6.18 Flattened Loop Formulation . . . 59
6.19 Natural Loop Formulation . . . 59

# 1 Introduction

Finite Element Methods (FEM) is a numerical method for finding approximate solutions to partial differential equations. It is used for modeling physical systems including solid structures, fluid mechanics, electromagnetic fields and many others [6]. These applications make FEM ubiquitous in science and engineering and one of the key applications of high performance computing. In FEM the (normally 2D or 3D) domain is broken up into a 'mesh' of many, possibly non-uniform, elements, hence the name "Finite Element Methods".

There are three stages in solving a partial differential equation with FEM. First, we compute the result of a 'kernel' for each element of the mesh, next we assemble these local solutions into a system of global equations and finally we solve this system. These stages are respectively named 'local assembly', 'global assembly' and 'solution'.

Local assembly is often the most time consuming stage of FEM. The exact structure of the local assembly 'kernel' is highly dependent on the problem and the mesh, but does have a characteristic form: a deep loop nest in which each loop has a low trip count and the innermost loop contains a complex expression. Local assembly is extremely data-parallel, typically each element and each iteration of the loop nest can be computed independently.

Historically programs have been hand-written in a low-level language in order to apply FEM to a specific problem. Although this allowed local assembly to be hand optimised for the specific problem, in general it was an unfortunate and expensive duplication of effort. The problem is especially severe since previous research[20] has shown that the best optimisation strategy is highly dependant on the target architecture and the parameters of the problem - optimisations were not portable between problems or between architectures and changing either of these things required a rewrite or at least creating and maintaining distinct backends. This has led to the creation of frameworks which accept a high level description of the partial differential equations for a specific problem along with a mesh, and which then produce optimised code for that problem on a target architecture. Since optimisations have a direct scientific payoff (faster simulation means the simulation can be made larger and more accurate), designing a composable set of optimisations that can be applied to specific FEM problems and architectures within these frameworks is an active area of research.

There is ongoing work to exploit the parallelism of local assembly on CPUs within these frameworks using COFFEE[1] [17, 18]. COFFEE is a compiler that manipulates local

[1] COmpiler For Fast Expression Evaluation

assembly kernel Abstract Syntax Trees (ASTs) to systematically apply various optimisations. These optimisations take advantage of the specific structure of the kernel to increase instruction-level parallelism and register locality.

Graphical Processing Units (GPUs) have an order of magnitude more scope to take advantage of data parallelism since they can execute thousands of operations at once whereas even multicore CPUs using vector operations can only execute tens of operations at once (albeit at lower latency). GPUs' novel architecture also offers new avenues for optimisation, for example by exploiting their fine-grained memory hierarchy. Local assembly's highly parallel nature makes it an excellent candidate for running on a GPU.

Previous work[2, 3, 8, 19] has investigated using GPUs to accelerate local assembly for FEM with good results. However this has generally been restricted to only a few specific problems. To our knowledge no codes exists that will produce optimised local assembly kernels targeting GPUs for a broad range of unseen problems.

The opportunity is clear: FEM is a critical application of HPC, ubiquitous in science and engineering, and local assembly is one of its most time consuming phases. Improving local assembly for a specific problem would let us improve the accuracy of that simulation but improving local assembly in Firedrake allows everybody[2] to get the benefit immediately for free. As for GPUs they are, theoretically, the perfect architecture for FEM but the difficulty involved in programing GPUs has meant that their adoption for FEM has been sporadic even among tools such as Firedrake where this cost would only have to be paid once. If we can demonstrate that the potential benefits of using GPUs for local assembly are as large as we imagine we can begin to justify the cost of solving the remaining problems which stand in the way of performing FEM entirely on the GPU.

## 1.1 Objectives

The aim of this project is to investigate optimisations to improve the performance of local assembly in Firedrake on GPUs. We use CUDA as our tool chain of choice and evaluate the performance of our optimisations on a wide range of benchmarks to ensure our optimisations will generalise well to unseen FEM kernels. We measure the performance of our optimisations on a NVIDA GRID K520.

We systematically investigate four optimisations: using constant memory to store basis functions, tuning the parameters of a kernel launch, using multiple threads per element and loop unrolling. The choice of optimisations is justified in chapter 5. Our investigation is described fully in chapter 6. Chapter 2 briefly describes FEM, the Firedrake tool chain and the GPU programming model. The methodology we use to evaluate the optimisations is described in detail in chapter 4. Finally, in chapter 7 we evaluate some of these optimisations and the success of this project.

[2] Or at least those who are Firedrake users.

## 1.2 Contributions

This thesis systematically investigates optimisations to improve the performance of local assembly in FEM for GPUs. We do this in the context of a code generation scheme for FEM in which, having implemented these optimisations once, they can be applied to a huge number of FEM problems. To our knowledge such an investigation has not been previously attempted.

The main contributions of this thesis are as follows:

- We show that storing the basis functions in local memory is catastrophic to the performance of problems with $>$ 1st degree basis functions, and that using constant hoisting to move the basis functions into constant memory can produce speedups of up to eighteen times. We produce patches allowing this optimisation to be immediately incorporated into Firedrake and PyOP2.
- We investigate parameter tuning to improve the occupancy of the generated CUDA kernels. We find that PyOP2's current strategy for choosing the three critical parameters which effect occupancy: registers per thread, blocksize and elements per block is not optimal for any of the benchmarks studied. We show that better choice of these parameters can result in a threefold performance improvement in some cases and that while the best results (1.93 times average speedup) are only achieved by tuning the parameters individually for each problem we can choose a single set of parameters to achieve an average speedup of 1.80 times and improve the performance of all but one benchmark.
- We examine the effect of forbidding and encouraging loop unrolling using `pragma unroll`. We discover that while loop unrolling is critical to the performance of kernels with low arithmetic intensity only in a minority of cases where the loop trip counts are large enough that CUDA is reluctant to unroll them by default does encouraging CUDA to unroll them improve performance and this improvement in of the order of 10 to 30%. We also show that CUDA frequently ignores the pragma especially for nested loops with large trip counts.
- We present a novel technique for allowing any number of threads up to one a limit of one per local matrix entry to corporate on assembling an element. The technique flattens the $j$ and $k$ loops and assigns sections of the flattened loop to each thread. We show that for this technique to be effective the number of threads per element must be a must be a divisor of the tensor size and close to a divisor of the block size and we show the necessity of coalesced memory accesses in this scheme. We show that combining this approach with parameter tuning improves the performance of low order and low arithmetic intensity kernels.
- Finally we consider the optimisations together and show average improvements of 13 times rising to 35 times for some benchmarks. We also show we can achieve an average 57% of peak FLOPs on the NVIDA GRID K520.

- Each of the above results is validated through testing with three separate problems of greatly varying complexity, three element types including 2D and 3D elements, polynomial basis functions of degree 1-4 and the combinations thereof. This provides evidence that these results will generalize to yet unseen FEM problems: one of the key benefits of Firedrake.

# 2 Background

This chapter will give a brief overview of Finite Element Methods

## 2.1 Finite Element Method

A Partial Differential Equation has the form:

$$L(u) = f \tag{2.1}$$

To apply FEM we first derive a *weak form* of the equation by multiplying by a *test function* $v$ and integrating over the domain.

$$\int_\Omega L(u)v \,\mathrm{d}X = \int_\Omega fv \,\mathrm{d}X \tag{2.2}$$

Intuitively a solution to the weak form is an approximate solution to the strong form. Next we discretise $u$ in terms of a finite number of *basis functions*. The choice of basis functions affects the accuracy of the solution.

The domain is divided into an unstructured mesh of elements. Normally each element has the same number of vertexes.

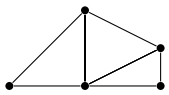


Figure 2.1: A 2D unstructured mesh with triangular elements

Solving the weak form involves three stages: local assembly, global assembly and solution. In local assembly we compute a matrix $\mathbf{M}_i$ and a vector $\mathbf{b}_i$ for each element $i$ in the domain. $\mathbf{M}_i$ has dimensions $N_e \times N_e$ and $\mathbf{b}_i$ has length $N_e$ where $N_e$ is the number of vertices in each element. This computation involves using Gaussian quadrature to evaluate integrals at each element.

Next we assemble these local matrices and vectors into a sparse global matrix $\mathbf{M}$ and vector $\mathbf{v}$. This is global assembly which combines the contributions from each element.

Finally we solve $\mathbf{Mx} = \mathbf{v}$ for $\mathbf{x}$ which gives us the solution.

### 2.1.1 Local Matrix Approach vs. Addto Algorithm

Local assembly is an integration over quadrature points for each entry in the local matrix and global assembly computes each non-zero entry by summing contributions from a subset of the local matrix entries. This suggests an alternative strategy: rather than performing assembly in these two phases we could perform the integration over quadrature points and add these contributions directly into the correct place in the global matrix.

This strategy, which is known as the *addto* algorithm is the more traditional approach and [20] shows that it is the optimal approach on multi-core architectures like modern CPUs. However, [20] also shows that that the *Local Matrix Approach* (LMA) - the approach we originally described, which explicitly constructs the local matrices - is optimal in the case of many-core architectures like modern GPUs. The downside of the Addto algorithm, particularly on GPUs where we must expose a large amount of parallelism to achieve good performance, is twofold: firstly, concurrent updates to the same entries in the global matrix can cause data races, necessitating expensive atomic operations or using coloring to avoid assembling conflicting elements at the same time. Secondly, the global matrix is large and sparse, so is normally stored in a compressed sparse row (CSR) format. Under this format accessing an entry requires searching the sparsity structure of the matrix, causing control flow divergence and uncoalesced accesses (see section 2.5). These problems make LMA preferable to the Addto algorithm on GPUs despite duplicating computation.

Finally, we note that while the Addto algorithm explicitly constructs the global matrix, the Local Matrix Approach could avoid this entirely and instead proceed directly to the solving phase, a 'matrix-free' approach.

As described in [24], PyOP2 uses the LMA on GPUs and the Addto algorithm on CPUs. The distinction between these approaches will be relevant in chapter 3 where we describe prior work on optimising local assembly on GPUs.

### 2.1.2 Local Assembly Structure

As mentioned above local assembly involves computing a local assembly kernel for each element in the mesh. Local assembly kernels normally contain a three loop nest, two loops over the local basis functions and an inner loop over the quadrature points. Listing 2.1 shows the structure:

Listing 2.1: Loop nest structure of a local assembly kernel

```
1 for (element in elements) {
2     // Jacobian
3     for (int j=0; j<J; j++) {
4         for (int k=0; k<K; k++) {
5             for (int ip=0; ip<IP; ip++) {
6                 // Code
7                 A[j][k] += ...;
```

```
8                 }
9             }
10        }
11    }
```

### 2.1.3 Meshes

As described above we must discretise the space in which we are interested into a finite number of elements. Together these elements are known as the 'mesh'. The space (and hence the elements) may be one-, two-, three- or higher-dimensional. Typically one-dimensional meshes are only used for didactic purposes, the vast majority of practical applications use two or three dimensional meshes. Higher-dimensional meshes are possible but more unusual. Normally every element in a mesh has the same number of vertices, although FEM does not require this. Typical shapes for elements are triangles or quadrilaterals in 2D meshes and tetrahedra or cuboids in 3D meshes.

Meshes may be 'Structured', 'Unstructured' or 'Semi-Structured'. In structured meshes we can compute the array indices of mesh elements directly - we can access this data with only a single array access `A[i]`. Unstructured meshes require a level of indirection - `A[B[i]]`. Semi-structured (also known as extruded) meshes have an unstructured base mesh, which requires an indirect access, but each element of the base mesh is also associated with a column of other elements which, after this initial indirect access can be accessed directly. One of the key advantages of FEM is that it can be used with fully unstructured meshes. This thesis investigates only unstructured meshes.

### 2.1.4 Basis Functions

The choice of basis functions has a large impact on the solution and the computation. Piecewise linear basis functions and Lagrange polynomial basis functions are common choices although more exotic basis functions are possible.

### 2.1.5 Jacobian

Rather than recompute the basis functions for each element it is more convenient to map each element onto a standard reference element. In practice to achieve this we compute the determinant of the Jacobian matrix for each element, the Jacobian determinant is a scaling factor that relates the differential area of the element to that of the reference element.

Finite Element Methods are a large topic and this overview has been extremely brief, concentrating on the part which is important for our purposes: local assembly. For a complete treatment see [10].

Listing 2.2: Example of UFL

```
from firedrake import *

degree = 1

mesh = UnitCubeMesh(20, 20, 20)
V = FunctionSpace(mesh, "CG", degree)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Function(V)

a = f * u * v * dx

A = assemble(a)
A.M
```

## 2.2 Firedrake Toolchain

Firedrake[23] is a new tool for automatically finding numerical solutions to partial differential equations using FEM. A user specifies the weak form of the PDE, selects the appropriate basis functions and defines the mesh at a high level using Unified Form Language (UFL) (see section 2.2.1). Firedrake then lowers this description by generating appropriate kernels using a modified version of FFC (see section 2.2.2). These kernels are passed to PyOP2 (see section 2.2.3), which executes them in parallel over the mesh to generate the global matrix and vector. Finally PETSc is used to solve this system of linear equations.

Firedrake performs each of these steps at runtime. A user can execute the high level UFL code directly and Firedrake will lazily generate and execute optimised code for solving the PDE, targeting to the current platform.

### 2.2.1 UFL

Unified Form Language (UFL) [1] is a Domain Specific Language (DSL) used for expressing variational formulations of partial differential equations. It allows a user to specify the solution to a PDE at a very high level in a pseudo-mathematical form. The power of UFL is that it does not commit to any specific implementation of the solution: the same abstract UFL problem specification can be lowered into a concrete form by many different *form compilers*. This has many advantages, it reduces the cost of solving new PDEs, it means that solutions are portable between any platform that uses UFL and it provides a separation of concerns - those who need to solve new PDEs do not necessarily have to know or worry about maintaining and improving a form compiler and vice versa.

### 2.2.2 FFC

Firedrake uses a modified version of FFC which was originally built as part of the FEniCS project. FFC takes the high level UFL description and produces C code which implements the numerical kernels. FFC itself calls FIAT to generate the arrays of data used in the kernel.

### 2.2.3 PyOP2

PyOP2[21, 24] is a performance-portable framework for applying numerical kernels in parallel to each element of an unstructured mesh. PyOP2 can target multiple backends including OpenMP, OpenCL on CPU, OpenCL on GPU and CUDA. When PyOP2 comes to run a kernel it passes the AST though COFFEE (see section 2.3), which generates optimised code for execution.

### 2.2.4 PETSc

Firedrake uses PETSc in the 'solution' phase to solve the system of linear equations.

## 2.3 COFFEE

COFFEE [17, 18] is a domain-specific compiler for optimising local assembly kernels. COFFEE accepts ASTs and generates C code including vector intrinsics. COFFEE applies optimisations including loop invariant code motion, padding, data alignment, loop interchange, loop unrolling and expression splitting in a systematic way to maximise register locality, instruction-level parallelism and SIMD vectorisation. Although some of these optimisations are preformed by vendor and research compilers, these compilers fail to achieve comparable results to COFFEE because they do not take full of advantage of the structure of local assembly kernels.

## 2.4 GPU Architecture

GPU architectures are typically 'many-core' in that they consist of a large number of very simple processing units, each of which has relatively few registers and a small cache. GPUs normally have a high-bandwidth connection to multiple banks of memory. To achieve good performance they rely on being able to hide the latency of accessing this memory by executing many instructions in parallel. The GPU device operates under the control of a host Central Processing Unit (CPU). The host can transfer data to and from the GPU and can also start computations.

## 2.5 GPU Programming Models

This section describes the two dominant GPU programing models, CUDA and OpenCL. Our investigation deals exclusively with CUDA so the description of OpenCL is necessarily brief and only relates the OpenCL concepts to their equivalent concepts in CUDA.

### 2.5.1 CUDA

CUDA code is compiled with `nvcc`, a proprietary compiler developed by NVidia. CUDA code broadly has C syntax and semantics extended with extra keywords to identify GPU functionality. For example, prefixing a function declaration with `__global__` marks it as a 'kernel' function. In GPU programming a 'kernel' is a function that is called from the host (the CPU) but executed on the device (the GPU). Similarly `__device__` marks the function as callable and executable only on the device and `__host__` marks it as callable and executable only on the host.

GPU programing typically follows a Single Instruction Multiple Threads (SIMT) model[1], this is halfway between Simultaneous Multithreading (SMT) which allows the simultaneous execution of many completely separate instruction streams and Single Instruction Multiple Data (SIMD) which has a single rigid instruction stream but each instruction can operate on a short vector of operands. In SIMT you write a function that operates on scalar values, as in listing 2.3, but when it comes to run this function on the GPU, thousands of threads run the code in parallel. Each thread gets a unique identifier which can be accessed by a variable (in CUDA the `(threadIdx, blockIdx)` pair is unique). These identifiers are used to modify the behavior of each thread - in listing 2.3 the variables are used to calculate a unique index into the arrays for each thread.

Listing 2.3: CUDA Kernel for adding vectors

```
1 __global__ void add(float *A, float *B, float *C) {
2     int i = threadIdx.x + blockIdx.x * blockDim.x;
3     C[i] = A[i] + B[i];
4 }
```

The CUDA terminology for running a function on the GPU this way is to "launch a kernel". In CUDA, threads are arranged into 'blocks' and 'blocks' are arranged into 'grids'. A kernel is associated with a single grid, each grid may contain many billions of blocks (although only a few will execute on the GPU at the same time[2]). Each block groups a maximum of 1024 threads together. Both grids and blocks may be 1-, 2- or 3-dimensional. Each block of threads shares some fast memory referred to as "shared memory" and threads within a block can synchronise by calling `__syncthreads()` this allows threads within a block

[1] The term was coined in NVidia documentation but also applies to OpenCl.

[2] The maximum number of simultaneously resident blocks is a small multiple of the number of Streaming Multiprocessorss (SMs) - on recent architectures the multiple is 32.

to cooperate. It is not possible to synchronise threads from different blocks. Higher level synchronisation can only be achieved by waiting for kernels to complete their execution. If any thread in a block reaches a `__syncthreads()` statement then every thread in the block much reach that statement, code that does satisfy this property is 'barrier divergent'. The behavior of barrier divergent code is undefined. Within each block, threads are grouped into 'warps' of 32 threads[3]. A warp is guaranteed to execute in lock step.

Launching a kernel has a special syntax, we can launch the `add` kernel like so `add<<<gridDim,blockDim>>>(A, B, C)`. The arrangement of threads is specified by the `gridDim` and `blockDim` variables. Unsurprisingly `gridDim` determines the dimensions of the grid and `blockDim` determines the dimensions of the blocks, if $g_x \times g_y \times g_z$ is the grid dimension and $b_x \times b_y \times b_z$ is the block dimension then the total number of blocks is $g_x g_y g_z$, the number of threads in each block is $b_x b_y b_z$ and the total number of threads is their product.

The memory hierarchy described in 2.4 is explicitly managed: you must choose where each variable is stored by prefixing its declaration with a keyword, for example `__local__` to cause that variable to be stored in local memory. This affects access latency but also visibility to other threads. There is only one instance of any memory declared global which every thread can access. Local memory is per-thread and private to that thread, while shared memory is, as mentioned, per-block. This is in contrast to memory management in most CPUs, when you can explicitly choose to move data between registers and main memory but exert only very indirect control over the L1, L2 and L3 caches, for example by issuing cache flush instructions or by using prefetch hints.

When a warp accesses global memory the GPU attempts to 'coalesce' these accesses into fewer transactions. For example, a 32bit load at address 128 and a 32bit load at address 160 might be coalesced into a single 64bit load from address 128. This is transparent to the application except insofar as un-coaleased accesses suffer significant performance penalties. Exactly which memory access patterns can be coalesced depends on the architecture but in general the loads must be aligned and sequential.

When a warp encounters a branch and some threads take the branch but others do not we have 'warp divergence' also known as branch divergence or control flow divergence. Warp divergence hurts performance because warps execute in lock step, if any thread in a warp takes a branch then the whole warp has to execute each instruction in that branch with any threads which would not have taken the branch disabled. For the purposes of intuition we can imagine warps being implemented as 32 lane SIMD instructions where each lane can be separately predicated.

The most import consideration to achieve good performance on a GPU is to have enough fine-grained parallelism [20, 22] to take advantage of the GPU's highly parallel nature. However, the following issues can also have a large impact on the performance of GPU code:

[3] According to [22] "The term warp originates from weaving, the first parallel thread technology."

- Minimise branch divergence within a warp.
- Access global memory in a way that allows coalescing.
- Maximise utilisation by maximising block occupancy and/or by providing sufficient instruction-level parallelism.
- Minimise host-device data transfer.

### 2.5.2 OpenCL

Our investigation deals with CUDA exclusively so we will not go into so much detail about OpenCL's programing model. Suffice to say, it is broadly similar to CUDA's. In simple cases such as listing 2.4 the syntax is essentially identical to CUDA except for substitutions of the various keyword in table 2.1. We will use CUDA terminology throughout this report but translations can be found in table 2.2.

The most important difference between CUDA and OpenCL is that while OpenCL is an open standard which principally targets GPUs (but also CPUs and more exotic architectures), CUDA is a proprietary invention of NVIDA which only targets NVIDA GPUs. Although OpenCL targets many more platforms, CUDA's documentation and tooling is more mature. Since our aim was largely investigative and duplicating every experiment for two programming models was of limited benefit but took significant extra work, we conducted our exploration with CUDA exclusively. Due to the close nature of the OpenCL CUDA programing models we expect the results of this investigation to be broadly applicable in the context of OpenCL.

Listing 2.4: OpenCL Kernel for adding vectors

```
1 __kernel void add(__global float *A,
2                   __global float *B,
3                   __global float *C) {
4     int tid = get_global_id(0);
5     C[id] = A[id] + B[id];
6 }
```

| CUDA | OpenCL |
|---|---|
| `__global__` | `__kernel` |
| `__constant__` | `__constant` |
| `__device__` | `__global` |
| `__shared__` | `__local` |
| `__device__` (function) | No annotation necessary |

Table 2.1: CUDA vs. OpenCL Keywords [5]

| CUDA | OpenCL |
|---|---|
| Threads | Work-item |
| Thread block | Work-group |
| Global memory | Global memory |
| Constant memory | Constant memory |
| Shared memory | Local memory |
| Local memory | Private memory |

Table 2.2: CUDA vs. OpenCL Terminology [5]

# 3 Related Work

Previous work has demonstrated significant speedups for FEM in general and for Finite Element assembly in particular on GPUs. However, this has almost exclusively been in the context of hand-tuning a specific implementation in order to achieve good performance. A representative example is work by Przemysław Płaszewski, Krzysztof Banas and Paweł Macioł [19] which compares CUDA, OpenCL and a single core CPU implementation of a specific "2D linear elastostatics problem" with curved quadrilateral elements. Their implementation assigns each element to a block and one or more tensor entries to each thread and uses the Addto algorithm. It achieves significant speedups.

Similarly, [7] Dziekonski et al present an optimised implementation of local assembly using CUDA for a specific FEM problem in computational electrodynamics. They consider curvilinear tetrahedral elements and vector basis functions up to polynomial order three, again assigning each element to a thread block.

As a final example, Komatitsch, Göddeke, Erlebacher and Michéa ([13], [12]) implement a high-order finite-element application for earthquake modelling on a cluster of GPUs using CUDA and MPU. They achieve speedups of up to 25 times, but again the problem is very specific and few lessons can be drawn about optimising finite element assembly on GPUs in general. Their application uses the Addto algorithm combined with element coloring to parallelise the computation, assigning a block to each element and a thread to each local matrix entry.

Work by Markall et al begun in [19] and extended in [21] show that the optimal strategy for assembly differs between multi-core and many-core architectures and specifically that LMA is a better strategy on GPUs as discussed in subsection 2.1.1. This work helps motivate a code generation approach to FEM. However, it is mostly concerned with the boundary between local and global assembly rather than how to optimise local assembly itself (although clearly the decision of LMA vs. Addto will have a large impact on which optimisations are effective).

The FEniCS project ([15, 16]) showed that a code generation approach to FEM has significant benefits and implemented some automated optimisations for Finite Element assembly but exclusively in the context of CPUs.

The Firedrake project ([21, 23, 24]), upon which this investigation is based also takes a code generation approach, but among other features extends support to GPUs via CUDA and OpenCL. In this context the LMA vs. Addto choice has been explored, favouring LMA on

GPUs, but no investigation into the optimal strategy for performing local assembly itself on GPUs has been made.

As part of Firedrake COFFEE [17, 18], an optimising compiler specifically for local assembly has been used to investigate a number of interesting optimisations including generalised loop invariant code motion, padding, data alignment, loop interchange, loop unrolling and expression splitting for local assembly on CPUs. This has improved the performance of Firedrake's CPU backend to the point where it outperforms the GPU backend.

Papers [3] and [4] by Cecka, Adrian and Eric come the closest to examining the problem of optimising local assembly in general on GPUs. They consider various strategies for assigning threads to an element: namely one thread per element, one thread per non-zero entry in the global matrix and one thread per row of the global matrix. They also consider various different methods for assembling the global matrix, for example assembling directly in global memory (the Addto algorithm) or computing each non-zero in local memory first then writing contributions out to the global matrix. The strategies were investigated for a single problem but for a variety of different polynomial degrees of the basis functions. They conclude that the best strategy for low-order elements is to assign one thread per non-zero entry per entry in the global matrix but for higher-order elements, it is better to perform assembly using one thread per element. This investigation was somewhat predicated on the necessity of explicitly producing the global matrix. Since later research showed LMA to be a better strategy in general we do not investigate these different strategies.

The existing work on optimising Finite element assembly on GPUs has shown that good speedups are possible for hand-tuned compared to optimised CPU implementations [7, 12, 13, 19] (it remains to be seen if this will continue to be the case in light of [17, 18]). It has also been shown that code generation techniques for FEM are powerful and competitive [15, 16, 21, 23, 24] on CPU as well as GPU. However, research into which optimisation strategies are useful in general for local assembly on GPUs is limited and what exists ([3, 4]) examines optimisations under a strategy for assembly which was later shown ([21]) to be suboptimal for GPUs. This work aims to begin addressing this gap by taking a systematic approach to investigating some of the optimisations applied by hand in the past.

# 4 Experimental Methodology

We now describe the methodology we used to evaluate the performance of changes to the Firedrake toolchain. A key benefit of Firedrake's approach is that it allows a huge variety of problems to be stated in a high level language. It is obviously impractical to test our optimisations on all of these problems, especially since there are (presumably) many important problems yet to be written, instead we follow the approach of [17], evaluating our optimisations on a range of representative benchmarks. These benchmarks are motivated below in section 4.3.

## 4.1 Timing Benchmarks

PyOP2 already includes robust instrumentation for measuring the runtime of various stages of the computation, specifically PyOP2 implements various timers which can be inspected after a run to see how long various sections of the code took to run. Two of these timers, `cuda_kernel` and `parloop_kernel`, are useful to us. `cuda_kernel` records how long the local assembly CUDA kernel takes to execute as measured by PyCUDA while `parloop_kernel` measures the time between the local assembly call being dispatched to the correct backend in PyOP2 and the result being returned, in the case of the CUDA backend this is effectively a thin wrapper which calls the CUDA kernel. Figure 4.1 shows that the times measured by the `cuda_kernel` and `parloop_kernel` are essentially identical with `parloop_kernel` being a tiny bit slower due to the Python wrapper overhead. We could use either timer for timing local assembly, in our experiments we use `cuda_kernel` since it does not carry this tiny overhead.

In order to get accurate timings we set two PyOP2 flags, setting `PYOP2_LAZY=0` forces evaluation of parallel loops to happen immediately rather than when the results are requested and `PYOP2_PROFILING=1` causes GPU kernels to be launched synchronously. In every case we run benchmark multiple times (normally 11) and discard the first result to avoid 'cold-cache' effects, for example the first time the PyCUDA executes the CUDA kernel it must compile the code and this effects the `parloop_kernel` timer. It is reasonable to discard the first result since in practice users run a small number of kernels repeatedly on a large amount of data rather than the other way around.

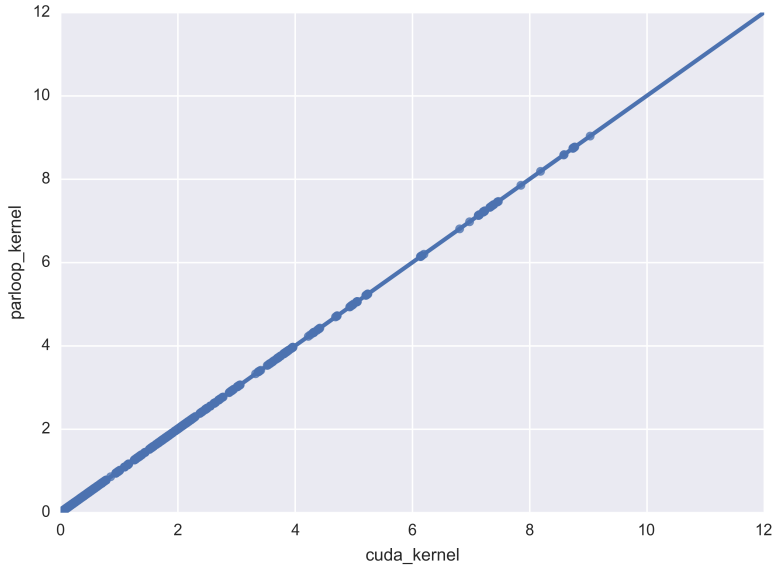



Figure 4.1: Times recored by `cuda_kernel` timer vs. those recored by `parloop_kernel` for every experiment conducted for this report. $y = 1.00027221902x + 0.000236040794433$

## 4.2 Profiling Benchmarks

We used two tools from the CUDA SDK, `nvvp` and `nvprof` to profile the benchmarks. Additionally we built a Python test harness to make it easy to conduct the necessary experiments

`nvprof` is a command line tool that can record various events and metrics of a CUDA application. It is invoked as follows: `nvprof [options] [application] [application arguments]`. Based on the options `nvprof` invokes the application with the application arguments (possibly multiple times), collecting statistics on it. We used `nvprof` is measure the number of double precision floating point operations carried out by the kernel in order to compute the FLOPs counts reported in chapter 7.

`nvvp` is effectively a visual frontend to `nvprof` which can either read log files produced by `nvprof` or invoke a CUDA application directly itself.

## 4.3 Selection and range of benchmarks

We chose three benchmarks to test our optimisations: 'mass', 'helmholtz' and 'elasticity'. These benchmarks are a representative sample of the equations used in real applications.

### 4.3.1 Mass

`mass` is the simplest possible benchmark. It performs the least amount of computation per element and represents a form of 'stress-test' for the optimisations, since the best strategies for local assembly when only a tiny amount of computation per element is done will differ considerably to that when there is a large amount of work per element.

Listing 4.1: Mass Benchmark

```
from firedrake import *

def mass(mesh, degree=1):
    V = FunctionSpace(mesh, "CG", degree)

    u = TrialFunction(V)
    v = TestFunction(V)
    f = Function(V)

    a = f * u * v * dx

    A = assemble(a)
    return A.M
```

### 4.3.2 Helmholtz

Helmholtz is a real-life kernel and a differential operator that is extensively encountered in scientific computing. The Helmholtz kernel can be used for imposing pressure in a compressible fluid and because of this it is frequently used in climate and ocean modeling.

Listing 4.2: Helmholtz Benchmark

```
from firedrake import *

def helmholtz(mesh, degree=1):
    V = FunctionSpace(mesh, "CG", degree)
    lmbda = 1
    u = TrialFunction(V)
    v = TestFunction(V)
    a = (dot(grad(v), grad(u)) + lmbda * v * u) * dx
    A = assemble(a)
    return A.M
```

### 4.3.3 Elasticity

Elasticity is another real life kernel which represents the other end of the complexity spectrum to mass, performing a significant amount of computation per element.

Listing 4.3: Elasticity Benchmark

```
from firedrake import *

def elasticity(mesh, degree=1):
    V = VectorFunctionSpace(mesh, 'CG', degree)
    u = TrialFunction(V)
    v = TestFunction(V)
    eps = lambda v: grad(v) + transpose(grad(v))
    it = inner(eps(v), eps(u))*dx
    A = assemble(it)
    return A.M
```

## 4.4 Benchmark Parameters

As well as the problem itself, a number of parameters have a large effect on the kernel which we will detail below. Throughout the remainder of this report the notation $\{\mathrm{p}, \mathrm{e}, \mathrm{degree}\ d\}$ refers to the benchmark consisting of problem $p$ with element type $e$ and polynomial degree $d$.

### 4.4.1 Polynomial Degree of Basis Functions

The polynomial order of the basis functions can be varied independently of the problem. In practical terms increasing the polynomial order increases the size of the basis functions, the size of the local tensor and hence the trip count of the $i$, $j$ and $ip$ loops. Sensible values for the polynomial order range from 1 to 8 or so, however Firedrake currently only supports low-order finite element methods of range 1 to 4. These will be the focus of our investigation. This is by no means a restriction, low order methods are extremely common and are used frequently in practice.

### 4.4.2 Mesh type

The shape of the mesh elements greatly effects the kernel, it obviously determines the number and size of the vertex coordinates which must be loaded from memory, but it also partly determines the size of the local tensor (and hence the trip count of the $i$ and $j$ loops) and the size of the basis functions, all of which has a profound effect on the performance characteristics of a kernel. Given this, it is important for us to consider a variety of element shapes, but even independently of this both two-dimensional and three-dimensional FEM techniques are important and we want to ensure our optimisations are effective for both.

As mentioned in section 2.1.3 the elements of a FEM mesh can take a variety of shapes but in practice elements are normally either triangles, in a two-dimensional mesh, or tetrahedra, in a three-dimensional mesh. In addition to these shapes we also consider meshes

with quadrilateral elements as these represent an intermediate step between triangles and tetrahedra, the quadrilateral meshes still have two dimensional vertex coordinates but they require more coordinates than triangle meshes and have larger local tensors than triangle meshes. Figure 4.2 shows the three types of mesh we consider.

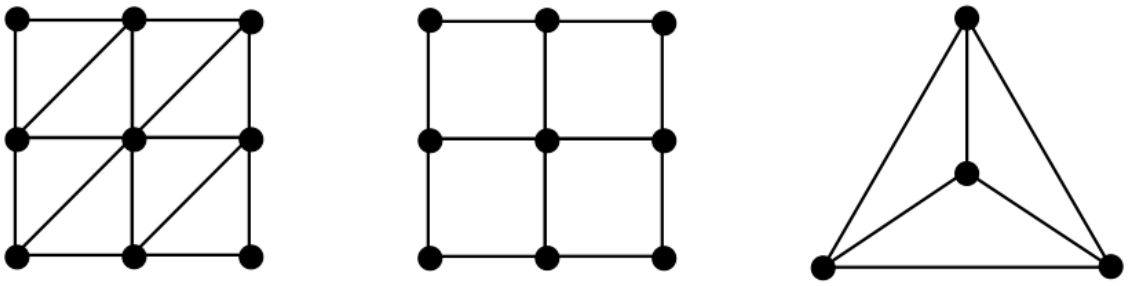


Figure 4.2: From left to right, a section of a 'triangle' mesh produced with, a section of a 'quadrilateral' mesh and a single tetrahedron element.

### 4.4.3 Mesh size

We would normally expect mesh size not to affect the performance of local assembly in the sense that runtime ought to increase linearly with the number of elements. In general we are interested in meshes which are as large as possible, however, the maximum mesh size we can fit in a given amount varies wildly with with the benchmark, mesh type and the polynomial degree of the basis function (since we must also store the local tensors).

We test with the largest mesh size that will fit in memory for each benchmark. the GRID K520 GPU has 4GB of global memory which translates to a meshsize between $10,000$ and $100,000$ depending on the benchmark.

We construct the meshes using helper methods from Firedrake located in `firedrake.utility_meshes`. What we refer to as the 'triangle' mesh is constructed with the `UnitSquareMesh` function (the mesh fills a unit square area but the elements are triangular), the 'square' mesh is again constructed with the `UnitSquareMesh` function but with the `quadrilateral` keyword argument set to `True` so that the mesh uses quadrilateral elements. The 'tetrahedron' mesh is constructed with the `UnitTetrahedronMesh` function.

`UnitSquareMesh` takes arguments `nx` and `ny` which specify the elements per side of the square. We choose values of `nx` and `ny` such that the total number of elements are equal to the desired mesh size and `nx` is as equal as possible to `ny` and the mesh is as square as possible.

Similarly `UnitTetrahedronMesh` takes arguments `nx`, `ny`, `nz` and we hand pick values for `nx`, `ny`, `nz` which produce meshes with the number of elements we desire.

### 4.4.4 Overview

The combination of these parameters results in 36 separate benchmarks detailed in table 4.1. Two of these {elasticity, tetrahedron, degree 4} and {helmholtz, tetrahedron, degree 4} produce kernels too large to be compiled by nvvc so we do not benchmark them. The parameters have a profound effect on the kernels being executed, for example the basis functions of the {mass, triangle, degree 1} benchmark (9 doubles) can fit the registers of a modern CPU whereas the basis functions of the {elasticity, tetrahdron, degree 4} benchmark (22680 doubles) can not fit in the L1 data cache, similarly the local tensor in the {mass, triangle, degree 1} benchmark is nine elements whereas the local tensor of the {elasticity, tetrahdron, degree 4} has 11,025 elements.

## 4.5 Correctness

It is easy to accidentally change the semantics of code when trying to optimise it. To avoid this we computed 'golden' expected values of the LMA matrix without our optimisations for each benchmark tuple and then as we developed the optimisations we constantly checked that the modified code still produced the correct results for each benchmark tuple. Figure 4.3 shows one such 'golden' matrix.

| Benchmark | Mesh Type | Degree |
|---|---|---|
| mass | triangle | 1 |
| mass | triangle | 2 |
| mass | triangle | 3 |
| mass | triangle | 4 |
| mass | quadrilateral | 1 |
| mass | quadrilateral | 2 |
| mass | quadrilateral | 3 |
| mass | quadrilateral | 4 |
| mass | tetrahedron | 1 |
| mass | tetrahedron | 2 |
| mass | tetrahedron | 3 |
| mass | tetrahedron | 4 |
| helmholtz | triangle | 1 |
| helmholtz | triangle | 2 |
| helmholtz | triangle | 3 |
| helmholtz | triangle | 4 |
| helmholtz | quadrilateral | 1 |
| helmholtz | quadrilateral | 2 |
| helmholtz | quadrilateral | 3 |
| helmholtz | quadrilateral | 4 |
| helmholtz | tetrahedron | 1 |
| helmholtz | tetrahedron | 2 |
| helmholtz | tetrahedron | 3 |
| elasticity | triangle | 1 |
| elasticity | triangle | 2 |
| elasticity | triangle | 3 |
| elasticity | triangle | 4 |
| elasticity | quadrilateral | 1 |
| elasticity | quadrilateral | 2 |
| elasticity | quadrilateral | 3 |
| elasticity | quadrilateral | 4 |
| elasticity | tetrahedron | 1 |
| elasticity | tetrahedron | 2 |
| elasticity | tetrahedron | 3 |

Table 4.1: Benchmarks

$$\begin{pmatrix}
0.01667 & 0.00833 & 0.00833 & & & & & & & \\
0.00833 & 0.05000 & 0.01667 & 0.01667 & 0.00833 & & & & & \\
0.00833 & 0.01667 & 0.03333 & 0.00833 & & & & & & \\
 & 0.01667 & 0.00833 & 0.05000 & 0.01667 & 0.00833 & & & & \\
 & 0.00833 & & 0.01667 & 0.05000 & 0.01667 & 0.00833 & & & \\
 & & & 0.00833 & 0.01667 & 0.05000 & 0.01667 & & 0.00833 & \\
 & & & & 0.00833 & 0.01667 & 0.05000 & 0.00833 & 0.01667 & \\
 & & & & & & 0.00833 & 0.05000 & 0.01667 & 0.01667 \\
 & & & & & 0.00833 & 0.01667 & 0.01667 & 0.05000 & 0.00833 \\
 & & & & & & & 0.01667 & 0.00833 & 0.05000
\end{pmatrix}$$

Figure 4.3: 'Golden' matrix `mass-unittriangle-10-1.npy`

## 4.6 Hardware

We test our benchmarks on a NVIDA GRID K520, its characteristics are listed in table 4.2.

| | **GRID K520** |
|---|---|
| **Chipset** | GK104 |
| **CUDA Compute Capability** | 3.0 |
| **Peak Double Precision** | 95 GFLOPs |
| **Peak Single Precision** | 2256 GFLOPs |
| **Peak Memory Bandwidth** | 192 GB/s |

Table 4.2: GPU Characteristics

# 5 Choice of Optimisations

We chose four optimisations to investigate:

- Constant hoisting (section 6.1)
- Parameter tuning (section 6.2)
- Loop unrolling (section 6.3)
- Multiple threads per element (section 6.4)

Of these, loop unrolling and multiple threads per element both aim to take advantage of intra-kernel parallelism while parameter tuning allows the GPU to make more efficient use of existing parallelism and 'constant hoisting' reduces the required number of memory accesses.

We chose to investigate intra-kernel parallelisation since taking advantage of intra-kernel parallelisation via vectorisation is one of the key techniques used by COFFEE. Additionally, using multiple elements per thread ought to increase the opportunity for coalesced memory accesses.

Parameter tuning is a classic GPGPU optimisation. In order to hide the latency of memory accesses many warps must be ready and waiting to be executed on each SM however the number of blocks that can 'fit' on an SM is limited by the resources of that SM. Parameter tuning allows us to adjust the use of these resources to increase the number of blocks which can fit on an SM at once.

Finally 'constant hoisting' was a pragmatic choice having observed that the basis functions could be very large in some kernels and also that they were ideal candidates to be stored in constant memory.

# 6 Investigation

This chapter describes our systematic investigation and evaluation of four optimisations: constant hoisting, parameter tuning, loop unrolling and multiple threads per element. For each optimisation we first discuss why implementing it ought to improve performance, we then explicitly state a hypothesis, suggest an experiment to test this hypothesis, detail the implementation of the optimisation and finally report and explain the results of the experiment.

## 6.1 Constant Hoisting

Effective use of bandwidth and the various levels in the GPU memory hierarchy are one of the most important factors that can limit GPU kernel performance. Although the programming model makes it look like each thread can access global memory completely independently, in practice memory accesses (like all operations) are vectorised - a whole warp must access memory together. When each thread in the warp accesses adjacent locations in memory this load or store is 'coalesced' and takes only a single memory access. Contrarily, if each thread accesses a different memory location the accesses must be serialised which is much less efficient. In addition to 'Global memory' NVIDA GPUs also have 'Constant memory'. Like Global memory, Constant memory is stored off-chip but unlike Global memory, Constant memory cannot be written by the kernel and is cached in a special constant cache. So long as every thread in a warp accesses the same location, reading from the constant cache is extremely fast.

In practice we can store a CUDA C variable in constant memory by prefixing its declaration with the `__constant__` qualifier. `__constant__` variables must be declared at the top level scope and are implicitly static. Listing 6.1 shows a typical use of the qualifier, in which constant memory is first initialised using `cudaMemcpyToSymbol` and then a kernel that uses that data is launched. Alternatively constant data can also be specified as a literal as in listing 6.2.

Listing 6.1: Example of `__constant__` qualifier

```
1 __constant__ double lookupTable[1000];
2
3 __global__ void kernel() {
4     // Use lookupTable here
5 }
6
```

```
7  void setupTable() {
8      double table[1000];
9      // populate table
10     cudaMemcpyToSymbol(...); // copy data from table to lookupTable
11 }
12
13 int main(int argc, char** argv) {
14     setupTable();
15     kernel<<<10, 32>>>();
16 }
```

Listing 6.2: Example of `__constant__` qualifier with literal data

```
1  __constant__ double lookupTable[1000] = {0.0, 0.1, ...};
```

### 6.1.1 Current Status

Currently PyOP2's CUDA back-end does not explicitly use constant memory at all. However, if we consider the different types of data a kernel accesses, the basis functions seem like excellent candidates to be stored in constant memory. Like the vertex coordinates, they are read only but unlike vertex coordinates every launch of kernel uses the same basis functions, so they only need to be transferred to the device once, the same locations are accessed repeatedly and all threads access the same elements of the basis functions in lock step.

### 6.1.2 Hypothesis

Storing the basis functions in constant memory should decrease the required memory bandwidth and, where this is the limiting factor, improve performance. This effect should be stronger on kernels with larger basis functions.

### 6.1.3 Experiment

To test this hypothesis we implemented 'constant hoisting' in PyOP2 and COFFEE such that any literal arrays in the kernel function are hoisted out of the kernel function and placed in constant memory and then ran our suite of benchmarks with and without this change.

### 6.1.4 Implementation

Listing 6.4 shows how the $\{\text{mass}, \text{triangle}, \text{degree } 1\}$ kernel can be transformed from its original form (see listing 6.3) to store the basis functions in constant memory.

Listing 6.3: Mass benchmark kernel

```
__device__ void form_cell_integral_0_otherwise(
    double A[1][1],
    double** vertex_coordinates,
    double** w0,
    int k, int j) {
  double FE0[4][4] = {
    {0.6220, 0.1666, 0.1666, 0.0446},
    {0.1666, 0.6220, 0.0446, 0.1666},
    {0.1666, 0.0446, 0.6220, 0.1666},
    {0.0446, 0.1666, 0.1666, 0.6220}
  };
  double W4[4] = {0.25, 0.25, 0.25, 0.25};
  double J[4];
  double K[4];

  // Calculate Jacobian
  J[0] = 0.5*(vertex_coordinates[2][0] + vertex_coordinates[3][0]
            - vertex_coordinates[0][0] - vertex_coordinates[1][0]);
  J[1] = 0.5*(vertex_coordinates[1][0] + vertex_coordinates[3][0]
            - vertex_coordinates[0][0] - vertex_coordinates[2][0]);
  J[2] = 0.5*(vertex_coordinates[6][0] + vertex_coordinates[7][0]
            - vertex_coordinates[4][0] - vertex_coordinates[5][0]);
  J[3] = 0.5*(vertex_coordinates[5][0] + vertex_coordinates[7][0]
            - vertex_coordinates[4][0] - vertex_coordinates[6][0]);
  double detJ = J[0]*J[3] - J[1]*J[2];
  K[0] = J[3] / detJ; K[1] = -J[1] / detJ;
  K[2] = -J[2] / detJ; K[3] = J[0] / detJ;

  const double det = fabs(detJ);
  for (int ip = 0; ip < 4; ++ip) {
    A[0][0] += (det * W4[ip] * FE0[ip][k] * FE0[ip][j]);
  }
}
```

Listing 6.4: Mass benchmark kernel after constant hoisting

```
__constant__ double FE0[4][4] = {
  {0.6220, 0.1666, 0.1666, 0.0446},
  {0.1666, 0.6220, 0.0446, 0.1666},
  {0.1666, 0.0446, 0.6220, 0.1666},
  {0.0446, 0.1666, 0.1666, 0.6220}
};
__constant__ double W4[4] = {0.25, 0.25, 0.25, 0.25};
__device__ void form_cell_integral_0_otherwise(
    double A[1][1],
    double** vertex_coordinates,
    double** w0,
    int k, int j) {
  double J[4];
  double K[4];
  J[0] = 0.5*(vertex_coordinates[2][0] + vertex_coordinates[3][0]
            - vertex_coordinates[0][0] - vertex_coordinates[1][0]);
  J[1] = 0.5*(vertex_coordinates[1][0] + vertex_coordinates[3][0]
            - vertex_coordinates[0][0] - vertex_coordinates[2][0]);
  J[2] = 0.5*(vertex_coordinates[6][0] + vertex_coordinates[7][0]
            - vertex_coordinates[4][0] - vertex_coordinates[5][0]);
  J[3] = 0.5*(vertex_coordinates[5][0] + vertex_coordinates[7][0]
            - vertex_coordinates[4][0] - vertex_coordinates[6][0]);
  double detJ = J[0]*J[3] - J[1]*J[2];
  K[0] = J[3] / detJ; K[1] = -J[1] / detJ;
  K[2] = -J[2] / detJ; K[3] = J[0] / detJ;

  const double det = fabs(detJ);
  for (int ip = 0; ip < 4; ++ip) {
    A[0][0] += (det * W4[ip] * FE0[ip][k] * FE0[ip][j]);
  }
}
```

We automated this transformation so it could be applied to any kernel by modifying COFFEE and PyOP2. In COFFEE we modified the `plan.ASTKernel.plan_gpu` function to search the AST for literal array declarations then hoist these declarations to the file-level scope and prefix them with the `__constant__` qualifier. This required small structural refactoring of COFFEE and PyOP2. PyOP2 has separate code paths for 'sequential' or CPU computation, OpenCL computation and CUDA computation but both GPU paths use the same `plan_gpu` code path in COFFEE which can transform the kernel to improve its performance before returning the modified kernel to PyOP2 for execution. There was also an implicilt requirement that COFFEE return code which conforms to the C99 standard from `plan_gpu` since PyOP2 uses `pycparser` to add the `__device__` prefixes to the kernel and `pycparser` accepts only C99 compliant code. This limits COFFEE's ability to take advantage of non-standard syntax/features on GPU platforms (for example the `__constant__` modifier).

We split the `plan_gpu` code path in COFFEE into two paths, one for CUDA and one for OpenCL to mirror the two code paths in PyOP2. Like PyOP2 we avoid duplication between these two code paths by implementing them as subclasses of a base `GPUPlan` class where the shared code can live. This matches PyOP2 implementation where the classes in `cuda.py` and `opencl.py` inherit from base classes in `device.py`.

We also removed the requirement for `pycparser` in this part of PyOP2 (`pyop2/cuda.py`) by moving the responsibility into COFFEE allowing COFFEE to return code which includes CUDA specific syntax.

### 6.1.5 Discussion

We ran all of the benchmarks described in chapter 4 with and without the constant hoisting optimisation. Figure 6.1 summarises the results as a speed up figure for each benchmark comparing the median result of the benchmark with constant hoisting to the median result without constant hoisting.

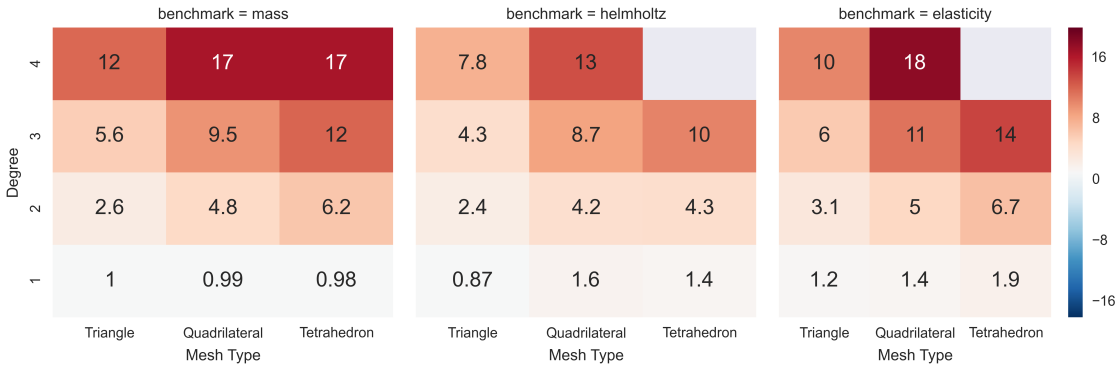



Figure 6.1: Constant hoisting speedup (compared to Firedrake's current implementation) for each benchmark

The plot consists of three subplots, each subplot shows a separate problem. Within each subplot the $x$ axis shows different mesh types while the $y$ axis shows different polynomial degrees. So each tile within a subplot shows a different benchmark and the color of that tile represents the speedup (or slowdown) for that problem. Benchmark combinations that we do not measure (see sub-section 4.4.4) are represented by gray tiles.

The performance of almost every benchmark improves with constant hoisting (31 of 34) and benchmarks with higher polynomial degrees and larger elements improve more than those with lower polynomial degrees or smaller elements. Of the remaining three, two become slightly worse (1-2%) and one becomes significantly worse (14%).

Figure 6.2 shows that the speedup for a benchmark increases as the size of the basis functions increase but that the scale of the increase varies significantly.

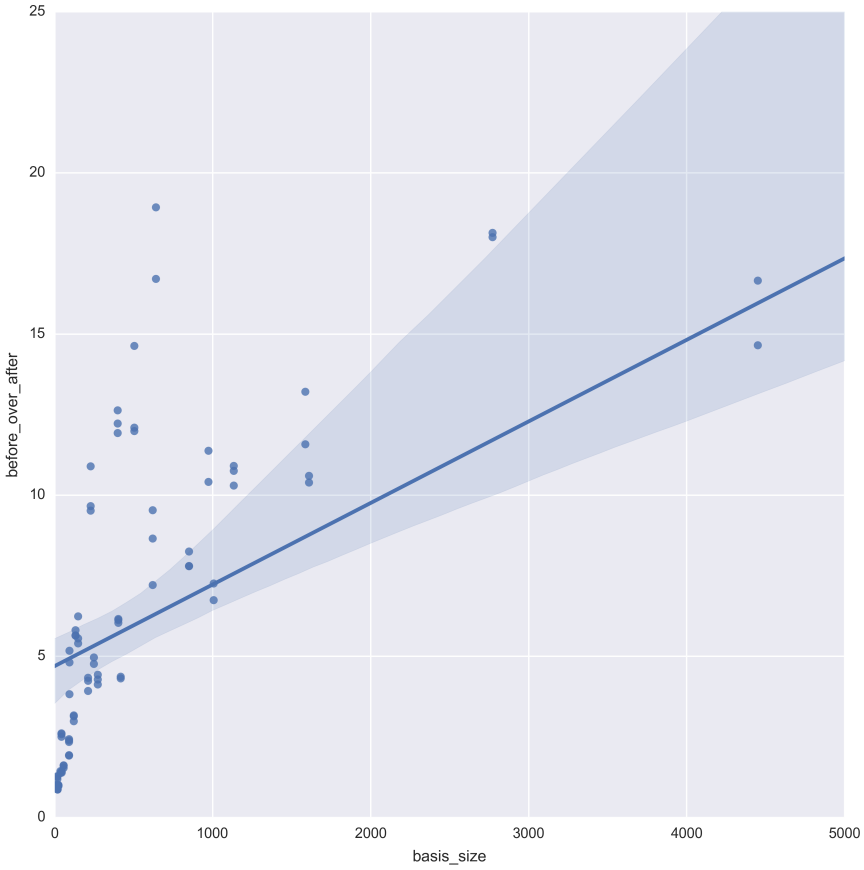



Figure 6.2: Constant Hoisting speedup vs. Basis Function Size

Profiling the memory bandwidth used by one of the most improved benchmarks, {mass, quadrilateral, degree 4 (16.70x), on a 10,000 element mesh with and without constant hoisting shows a large reduction in the number of local loads (94.1%), local stores (99.9%) and total device accesses (98.7%). We see no change to the number of accesses involving global and shared memory as we would expect since we do not modify the code that accesses that memory. We would expect to see an increase in the number of constant accesses, however CUDA profiling tools

do not support profiling the constant cache so we cannot measure this directly.

| | Baseline | Constant Hoisting | Percentage Decrease |
|---|---|---|---|
| **Local Load** | 7123313 | 420673 | 94.1 |
| **Local Store** | 121737492 | 67697 | 99.9 |
| **Shared Load** | 1599490 | 1599490 | 0 |
| **Shared Store** | 2164 | 2164 | 0 |
| **Global Load** | 2256147 | 2256147 | 0 |
| **Global Store** | 2250000 | 2250000 | 0 |
| **Device Total** | 240879253 | 3077616 | 98.7 |

Table 6.1: Memory Utilisation in $\{\text{mass}, \text{quadrilateral}, \text{degree } 4\}$ on a 10,000 element mesh

Examination of the assembly of the innermost loop of the original kernel shows it is storing the basis functions as immediates (listing 6.5), however after loading these values into registers it immediately stores them again in thread local memory and then much later loads them back from thread local memory in order to compute with them, this happens on every iteration of the $j$ loop. In comparison after constant hoisting (listing 6.6), the basis functions are stored in constant memory where they can be accessed directly. This explains the decrease in local memory accesses and suggests that constant hoisting improves the performance of most of the benchmarks because previously these kernels suffered from catastrophic register spilling due to the large basis functions which limited performance.

Listing 6.5: $\{\text{mass}, \text{quadrilateral}, \text{degree } 4\}$ assembly snippet

```
1 MOV32I R30, 0x27a1bd03;  // Load Immediate
2 STL.64 [R53+0x178], R24; // Store Local
3 MOV32I R31, 0xbfacc8fe;
4 STL.64 [R53+0x3d0], R24;
5 ...snip...
6 LDL.64 R24, [R57+0x78]; // Load Local
7 DMUL R12, R32, R2;      // Register Register Multiply
8 LD.E.64 R6, [R14];
9 DMUL R4, R28, R24;
```

Listing 6.6: $\{\text{mass}, \text{quadrilateral}, \text{degree } 4\}$ assembly snippet after Constant Hoisting

```
1 DMUL R10, R8, c[0x3][0xbb8]; // Register Constant Multiply
2 DMUL R26, R8, c[0x3][0xbc0];
3 DMUL R10, R10, R6;
4 DFMA R10, R10, c[0x3][0x20], R28;
```

Table 6.2 shows the benchmarks whose performance decreases a small amount. Examination of the assembly of these kernels show they already store the basis functions in constant memory, we have been unable to discover what causes this performance drop there is no notice diffrence in the profiling metrics and the instructions mix looks similar.

The performance of one benchmark, $\{\text{helmholtz}, \text{triangle}, \text{degree } 1\}$, significantly worsened after constant hoisting (shown in table 6.3). Examination of the assembly of this benchmark shows that although it already stores the basis functions in constant memory, after

| Problem | Degree | Mesh Type | Speedup |
|---|---|---|---|
| mass | 1 | triangle | 0.996664 |
| mass | 1 | quadrilateral | 0.987358 |
| mass | 1 | tetrahedron | 0.994096 |

Table 6.2: Benchmarks with small performance drops after constant hoisting

the constant hoisting is applied `nvcc` no longer chooses to fully unroll the $i$, $j$ and $ip$ loops, instead it only unrolls the $ip$ loop, this explains the drop in performance. Forcing `nvcc` to unroll this loop using `pragma unroll` (see section 6.3) improves the performance to 0.93 of the runtime without constant hoisting.

| Problem | Degree | Mesh Type | Speedup |
|---|---|---|---|
| helmholtz | 1 | triangle | 0.865738 |

Table 6.3: Benchmarks with large performance drops after constant hoisting

Constant hoisting improves performance for the majority of benchmarks and this performance increase is particularly dramatic on high order problems. The performance drop on a minority of benchmarks was small but noticeable it might be the case that we should not employ constant hoisting when the basis functions are very small but it will require further investigation to determine the exact conditions under which we should not employ constant hoisting.

The dramatic performance improvement was caused by a dramatic change in the memory access characteristics of the benchmarks and it did not seem sensible to continue our investigation ignoring this change. After all, optimisations that perform well in the context of a huge number of local memory accesses might perform very differently in the context of none. Due to this, in the remainder of this chapter we investigate the remaining three optimisations as changes to an implementation that includes constant hoisting and compare them to a new base line of the current implementation plus constant hoisting. Our final evaluation in chapter 7 compares the performance of our optimisations and reports the best speedups achieved against the original baseline.

## 6.2 Parameter Tuning

Another factor that strongly effects GPU performance is 'occupancy'. The occupancy of a streaming multiprocessor (SM) is defined as the number of active warps divided by the maximum possible number of active warps for that SM. By extension the occupancy of a kernel instantiation is the average occupancy of its SMs over the life of that kernel.

High occupancy allows an SM to hide the latency of accessing memory via context-switching to another active warp, preventing the SM from stalling. This context-switching is extremely fast since registers and shared memory do not need to be saved or restored,

rather these resources are allocated when an SM begins executing a block and are deallocated when every warp in the block has completed its execution. Each SM only has a finite number of registers and shared memory so these resources limit the number of blocks which can be concurrently executing on an SM - if a block requires 2KB of shared memory and each SM has only 4KB of memory then only two blocks can be active on the SM at once even if the SM's scheduling hardware supports four.

All other things being equal, increasing the occupancy of a kernel should only ever improve performance since the higher occupancy might provide an opportunity to context-switch and execute a warp where before an SM would have had to stall. Normally, however, increasing occupancy comes at the cost of compromising on one of the two resources, if a block uses fewer registers it might have to make more memory accesses and so the overall effect on performance could be negative. In practice if occupancy is very low ($\sim$10%) then increasing it (to say 30%) can dramatically improve performance but if occupancy is already high ($\sim$60%) then increasing it (to say 80%) may have little effect.

Medium or high occupancy is not the only way to hide the latency of accessing global memory. Alternatively, sufficient amounts of instruction-level parallelism could also hide latency as we describe in section 6.3.

Three variables principally affect a block's use of an SM's resources and hence a kernel's occupancy: threads per block or 'block size', shared memory per block and registers per thread. The value of each of these imposes an upper limit on the maximum possible occupancy. Whichever upper limit is lowest is the theoretical maximum occupancy of the kernel and the relevant variable(s) are the 'occupancy limiter(s)'.

Understandably maximum occupancy is a monotonically decreasing function with respect to both registers per thread and shared memory per block, as mentioned above registers and shared memory are limited resources on an SM - the more of them a block requires the fewer blocks can run on an SM simultaneously. The relationship between block size maximum occupancy is less straightforward, increasing the number of threads per block increases the number of registers needed for that block (since we are considering registers per thread) and so decreasing occupancy, but it will also increase the total number of warps in the block, possibly increasing occupancy.

CUDA provides mechanisms for controlling each of these three variables. Block size is specified at kernel launch time with the triple angle bracket syntax `the_kernel<<<gridSize, blockSize>>>([kernel arguments]);` or in PyCUDA via the `block` argument to the generated function object. Shared memory use is the sum of shared memory allocated statically by prefixing a variable declaration with `__shared__` and shared memory allocated dynamically at kernel launch time by passing a third optional argument in the triple angle bracket syntax: `the_kernel<<<gridSize, blockSize, sharedsize>>>([kernel arguments]);` or in PyCUDA by passing the `shared` argument to the generated function object. A kernel can access this memory by declaring an `extern` pointer: `extern __shared__ int *shared;`. Registers per thread

can be controlled either by the `-maxrregisters` `nvcc` flag or by specifying launch bounds for a kernel (see listing 6.7).

The `-maxrregisters n` flag forces `nvcc` to allocate a maximum number of $n$ registers to any one thread throughout the compilation unit. Specifying launch bounds provides more fine-grained but indirect control over register use, the two variables specified in the launch bounds are the `maxThreadsPerBlock` "maximum number of threads per block with which the application will ever launch" and (optionally) `minBlocksPerMultiprocessor` the "desired minimum number of resident blocks per multiprocessor", with these two variables `nvcc` derives an upper limit on the number of registers it should use per thread when compiling the kernel.

Listing 6.7: Example of launch bounds in CUDA C

```
1 __global__ void
2 __launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)
3 the_kernel(...) {
4 }
```

The CUDA SDK provides the `CUDA_Occupancy_Calculator.xls` spreadsheet which calculates the theoretical occupancy from the Compute Capability, total shared memory, threads per block, shared memory per block and registers per thread. Figure 6.3 shows the Occupancy Calculator {helmholtz, triangle, degree 4}

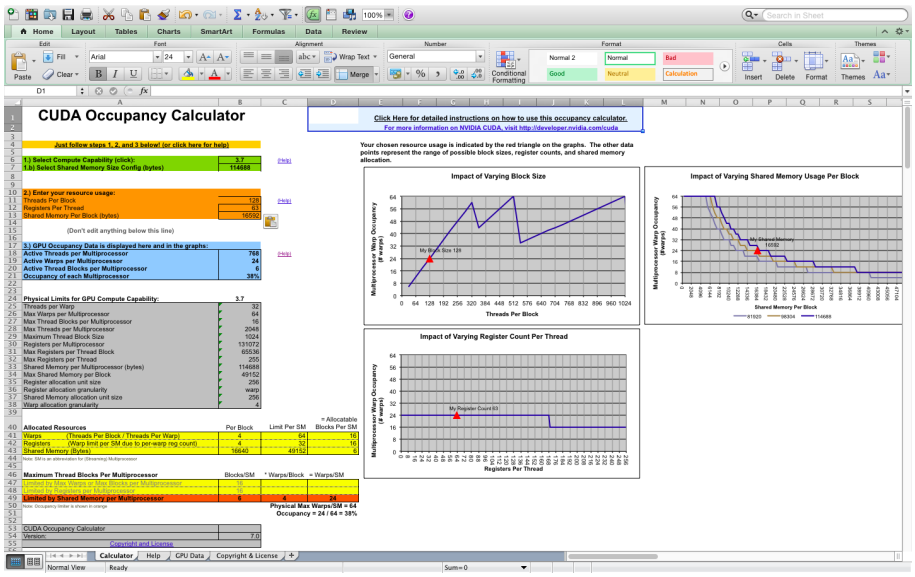



Figure 6.3: Screen shot of CUDAs Occupancy Calculator

### 6.2.1 Current Status

In terms of these three variables PyOP2 currently hard codes block size to 128 and gives `nvcc` control over how many registers to use, but the use of shared memory is more complicated. The principle use of shared memory in PyOP2's CUDA backend is to 'stage' the

vertex coordinates (and other indirect arguments of the kernel). These data are first loaded from global memory into shared memory then in a second step the local tensor entries are computed. Due to this, the amount of shared memory used by a block increases proportionally to the number of elements that block is assigned to compute. PyOP2 chooses the number of elements each block should compute, the partition size, to be the maximum number of elements whose indirect arguments could fit in the maximum allowable shared memory for a block in the worst case. In practice, the block does not need and is not allocated this much shared memory (the maximum possible) since normally some of the indirect arguments' values are shared between elements. For example, in a triangle mesh the majority of time a triangle will share its coordinates with other elements.

### 6.2.2 Hypothesis

PyOP2's fixed block size, lack of control over the number of registers per thread and its mechanism for determining partition size (and so indirectly shared memory per block) limits occupancy and hence performance.

### 6.2.3 Experiment

To test this hypothesis we perform a parameter sweep over block size, blocks per SM and partition size and benchmark each parameter set. If the hypothesis is true we expect some points in this parameter space to have higher theoretical and actual occupancy than the existing implementation and that these should be correlated with higher performance.

### 6.2.4 Implementation

We modified PyOP2 to allow these three variables to be set from the test harness. Rather than setting partition size directly we divide the original partition size by a factor. This maps nicely on to shared memory. Since the original partition size implies the maximum possible shared memory use, halving that partition size reduces the amount of shared memory used by half. This avoids problems due to the large differences between baseline partition sizes. Table 6.4 summarises the parameter space we searched.

| Parameter | Values |
|---|---|
| Block Size | 128, 256, 512, 1024 |
| Blocks per SM | 1, 2, 4, 8, 16 |
| Partition size factor | 1, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$ |

Table 6.4: Parameter space

### 6.2.5 Discussion

The experimental results show better choice of kernel parameters can significantly improve performance, up to 2.96 times in the case of $\{\text{helmholtz}, \text{triangle}, \text{degree } 3\}$ although some benchmarks were mostly unaffected. For example, the best choice of parameters for $\{\text{mass}, \text{tetrahedron}, \text{degree } 3\}$ only result in a 1.06 times speedup. PyOP2's current choice of parameters was not optimal for any benchmark.

Using the best choice of parameters for each benchmark results in an average speedup of 1.93 times (figure 6.5), however parameters which are good for one benchmark tend to also be good for other benchmarks as we can see from figure 6.4.

The parameter set $\{\text{blocksize} = 128, \text{partition size} = \frac{1}{2}, \text{blocks per SM} = 4\}$ has the best average speedup across all benchmarks, 1.80 times, this is very close to the 1.93 times speedup achieved by choosing the best parameters on a per benchmark basis suggesting that while the best performance is achieved by optimising the parameters for an individual program, significant improvements can be achieved just by optimising the parameters for a specific GPU.

From figure 6.4 we can see by far the most important parameter is the partition fraction which controls the number of elements asigned to each block and so indirectly with the shared memory per element. We can also see that as we increase blocksize our ability to modify the partition size without hurting performance goes down, this is sensible since assigning fewer elements to a block than the block has threads simply wastes resources. The least useful parameter by far to change is the number of blocks per SM which controls the register allocation.

We note that a one line change to PyOP2 to set the partition size to half its previous value would probably improve perfomance by 1.8 times.

## 6.3 Loop Unrolling

Loop unrolling is a well known and venerable optimisation technique. In the simplest case, unrolling a loop reduces the amount of overhead involved in keeping track of the induction variable at the cost of increasing code size, however it can also increase the amount of Instruction Level Parallelism (ILP) and this is particularly important in the case of GPUs where ILP is one of the two ways to hide the latency of accessing memory (the other is increasing occupancy, see secton 6.2).

CUDA C supports a pragma "`#pragma unroll`" to encourage `nvcc` to unroll a loop but reports on its efficacy are mixed. Table 6.5 summarises the NVIDA documentation on "`#pragma ↪ unroll`". Alternatively we can unroll the loops manually in the `jinga2` template or in COFFEE.


Figure 6.4: The average speedup (compared to the post-Constant Hoisting baseline) across all benchmarks for each parameter set.

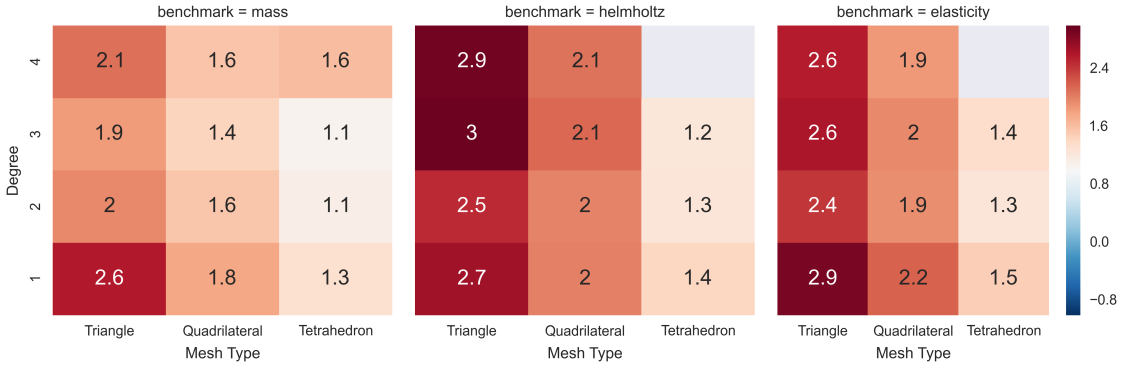



Figure 6.5: Maximum speedup compared to the post-Constant Hoisting baseline for each benchmark after a parameter sweep.

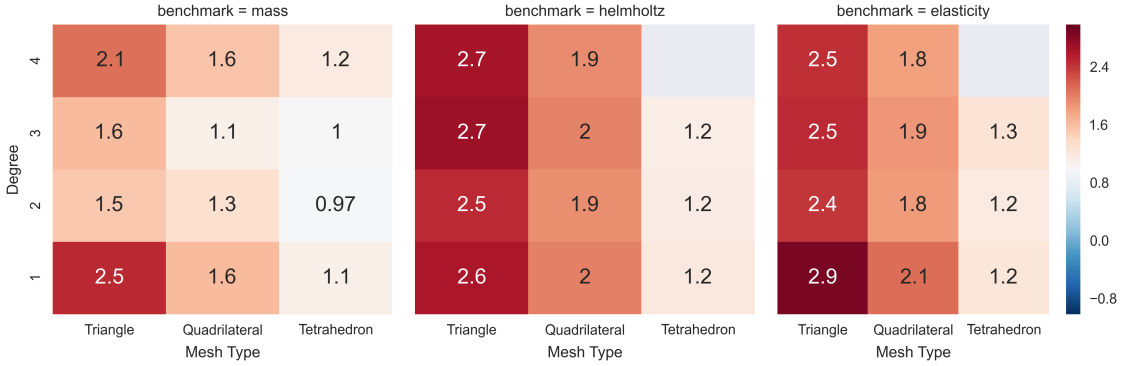



Figure 6.6: Speedup per benchmark for the parameters $\{\text{blocksize} = 128, \text{partition size} = \frac{1}{2}, \text{blocks per SM} = 4\}$

| Loop annotation | Result |
|---|---|
| No pragma | "By default, the compiler unrolls small loops with a known trip count" |
| `#pragma unroll` | if the loop has constant trip count it is completely unrolled |
| `#pragma unroll 1` | the loop is never unrolled |
| `#pragma unroll` $n$ | the loop is unrolled $n$ times |

Table 6.5: Use of `pragma unroll`[22]

### 6.3.1 Current Status

The code generated by PyOP2's CUDA back-end currently does not explicitly use loop unrolling either manually or via the unrolling pragma.

### 6.3.2 Hypothesis

Unrolling the $i$, $j$ and $ip$ loops or some subset of them should increase ILP and hence performance.

### 6.3.3 Experiment

We benchmarked the performance kernels with: no loop unrolled, with the $ip$ loop fully unrolled, with the $ip$ and $k$ loops fully unrolled and with all three loops unrolled. We used `pragma unroll 1` to force `nvcc` not to unroll a loop and `pragma unroll` to unroll a loop.

### 6.3.4 Discussion

Figure 6.7 shows the results of the experiment. If we consider columns associated with no unrolling we see significant performance drops, suggesting that `nvcc` automatically unrolls at least some loops. Examination of the assembly confirms this is indeed the case.

The performance drop for no unrolling is worst for $\{\text{mass}, \text{triangle}, \text{degree } 4\}$. This makes sense since this is the benchmark that has the highest number of executions of the $ip$ loop body relative to the number of operations within that loop so the effect of removing unrolling is strongest here.

Similarly the effect of removing loop unrolling is worse for $\{\text{mass}, \text{triangle}, \text{degree } x\}$ than for $\{\text{helmholtz}, \text{triangle}, \text{degree } x\}$ and worse again for $\{\text{elasticity}, \text{triangle}, \text{degree } x\}$ since `elasticity` has higher arithmetic intensity than `helmholtz` and `helmholtz` has higher arithmetic intensity than `mass`.

The same pattern can be seen between the different mesh types for the mass problem. As the tetrahedron benchmarks have higher arithmetic intensity than the triangle benchmarks, the performance drop of removing loop unrolling is higher for $\{\text{mass}, \text{triangle}, \text{degree } x\}$ than for $\{\text{mass}, \text{quadrilateral}, \text{degree } x\}$ and for $\{\text{mass}, \text{tetrahedron}, \text{degree } x\}$.

However while loop unrolling seems critical for the performance of the mass problem (removing it results at minimum performance drop of 9% and an average performance drop of 20%), it is much less important for the performance of the other problems (helmholtz has an average performance drop of 9% and elasticity an average of 3%) we suggest this is due to the significantly larger *ip* loop bodies of helmholtz and elasticity.

In six cases, using the unroll pragma at all, even to force no unrolling, is contraindicated: {helmholtz, tetrahedron, degree 2}, {helmholtz, tetrahedron, degree 3}, {elasticity, tetrahedron, degree 2} and {mass, tetrahedron, degree 1}. These cases could be caused by by the best unrolling strategy being to only partially unroll some loop or by `nvcc` choosing to unroll some combinations of loops we did not consider.

It is also clear from the results that `nvcc` does not always respect the pragma the results for {elasticity, tetrahedron, degree 3} are almost identical independently of how which loops we unroll despite the fact that there ought to be a 6000 times difference in the code size between unrolling all the loops and unrolling none. Examination of the assembly confirms this.

In a small number cases using pragma unroll increases performance on the order of 30% these seem to be cases where `nvcc` is reluctant to unroll the roll but will still unroll the loop if you ask it.

In general we conclude that while removing loop unrolling completely has a significant impact on some kernels loop unrolling alone it not sufficient to significantly improve the performance of local assembly. A more detailed investigation will require manual loop unrolling to avoid the issues we encountered.

| | Speedup | | |
|---|---|---|---|
| | Minimum | Mean | Maximum |
| No unrolling | 0.594476 | 0.918302 | 1.099113 |
| Unroll *ip* | 0.851797 | 0.974587 | 1.264343 |
| Unroll *ip*, *k* | 0.899732 | 1.026266 | 1.264882 |
| Unroll *ip*, *k*, *j* | 0.900355 | 1.042778 | 1.310426 |

Table 6.6: Summary of `pragma unroll` experiment

## 6.4 Multiple Threads per Element

As stated in chapter 2, local assembly kernels contain an embarrassing amount of parallelism. Out of the loops in a local assembly kernel, the `elements`, `i`, and `j` loops are completely parallel (the loop structure of a kernel is shown in listing 6.8).

Listing 6.8: Loop nest structure of a local assembly kernel

```
1 for (element in elements) {
2     // Jacobian
```

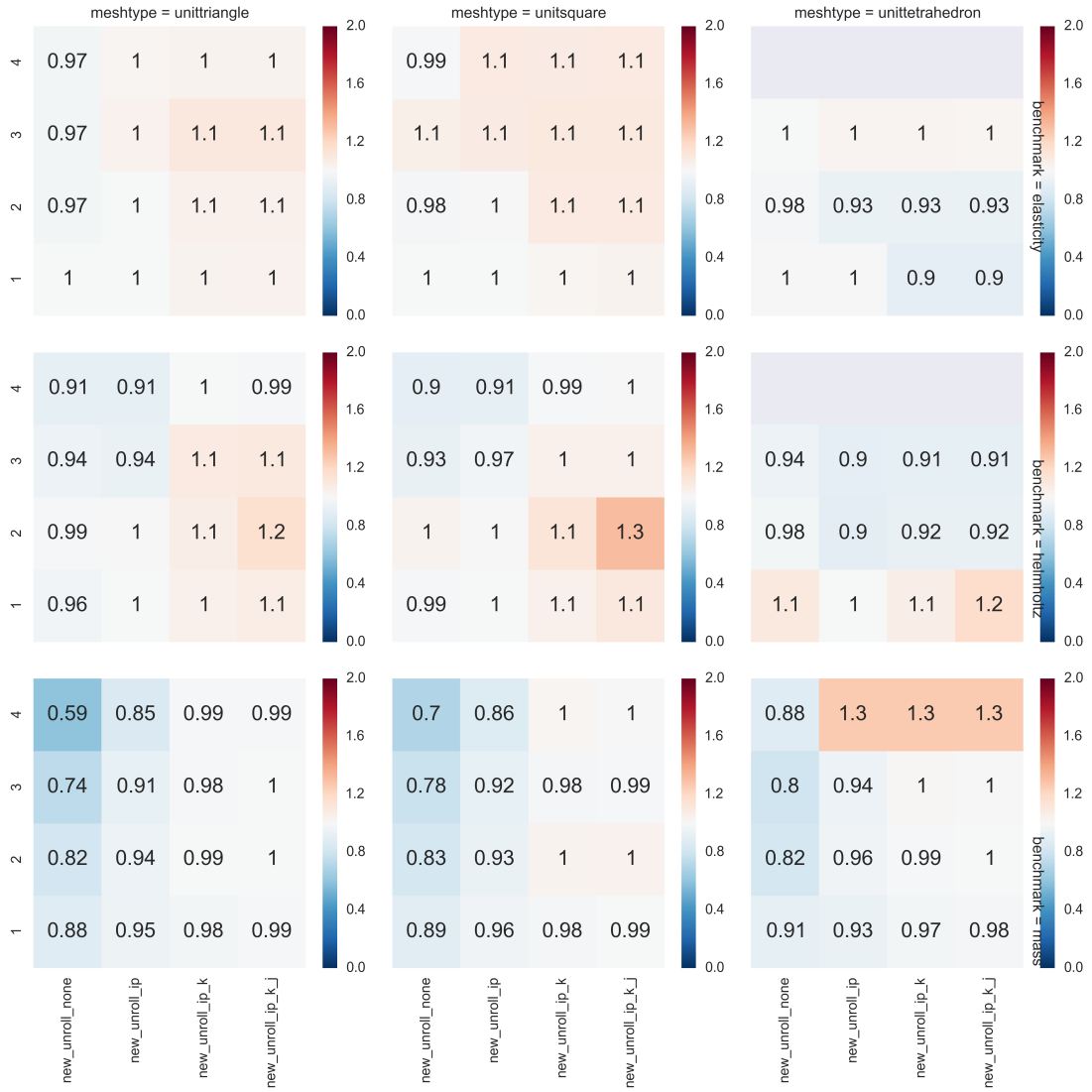



Figure 6.7: Speedup of unrolling no loops, unrolling only *ip* loop, unrolling *ip* and *k* loops and unrolling all three loops with pragma unroll, compared to post-Constant Hoisting baseline for each benchmark.

```
3       for (int j=0; j<J; j++) {
4           for (int k=0; k<K; k++) {
5               for (int ip=0; ip<IP; ip++) {
6                   // Code
7               }
8           }
9       }
10  }
```

PyOP2's current strategy for parallelising local assembly on GPUs uses only the parallelism found in the outer `element` loop. Specifically, PyOP2 breaks the elements into

groups and assigns each group to a block. Within that block of 128 threads[1] the first thread computes the local tensor of the first element, the second thread computes the local tensor of the second element, etc. After a thread finishes computing its current element it moves on 128 elements to find the next local tensor that needs computing, so if there are $e$ elements indexed from 0 to $e-1$, each thread computes the elements $E_i$ such that $i = k \times blockSize.x + threadIdx.x$. Figure 6.8 illustrates this strategy, showing which thread computes which local tensor entries in which elements by a number in that entry of the local tensor in that element. The character subscripts describe the ordering of the computations.

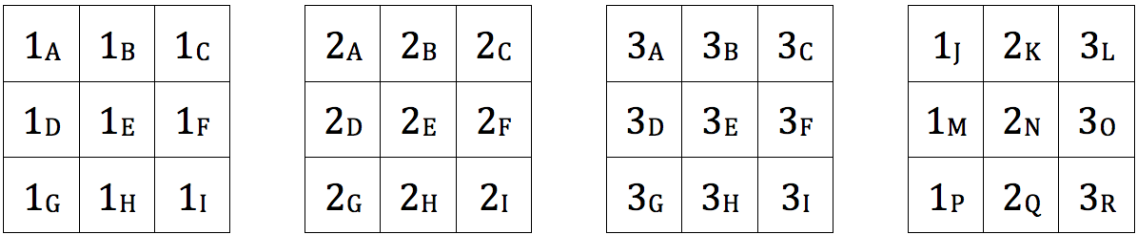



Figure 6.8: Current strategy for parallelisation

The kernel wrapper code which implements this strategy and is the entry point to our CUDA kernel is instantiated at runtime from jinja2 template (`PYOP2/pyop2/assets/cuda_indirect_loop.jinja2`). Listing 6.13 shows the critical section of the code for $\{\text{mass}, \text{quadrilateral}, \text{degree } 4\}$.

This strategy ignores the parallelism in the $i$ and $j$ loops but this does not necessarily hurt performance so long as the mesh is large enough that it still exposes sufficient parallelism and the GPU can make efficient use of this parallelism. Typically applications need to launch tens of thousands of threads to make full use of the GPU's SMs so this approach may work well on large meshes.

Figure 6.9 and figure 6.10 illustrate alternative strategies in which multiple threads cooperate to compute the local tensor of a single element.

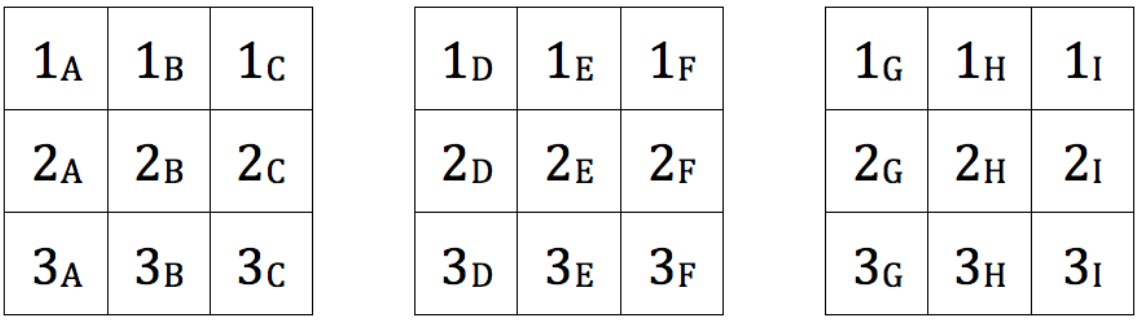



Figure 6.9: Chunked strategy for parallelisation

We implemented both of these alternative strategies in PyOP2 so that an arbitrary number of threads (up to the limit of one thread per local tensor entry) can work on the same ele-

[1] This block size is hard-coded as $(128, 1, 1)$.

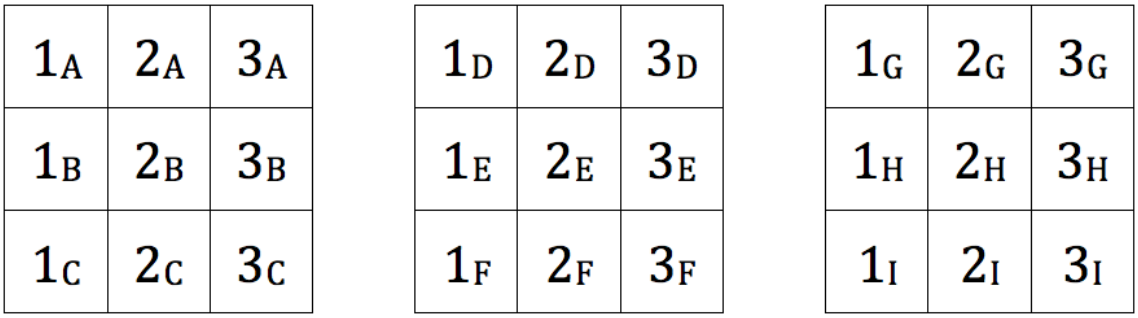



Figure 6.10: Coalesced chunked strategy for parallelisation

ment. This involved changing `PYOP2/pyop2/cuda.py` and `PYOP2/pyop2/assets/cuda_indirect_loop.jinja2` so that the kernel wrapper template takes an additional parameter, `threads_per_local_array`, and transforms the *element*, $i$ and $j$ loops.

Our implementation works by compressing the *element* loop so multiple threads are assigned the same element, then flattening the $j$ and $k$ loops[2] into a single new loop ($e$) and assigning portions of this new loop to each thread. Within this new flattened loop we then reconstruct the induction variables of the $j$ and $k$ loops from the induction variable of the $e$ loop.

Listing 6.10 shows loop flattening applied to a three loop nest while listings 6.11 and 6.12 show two strategies for parallelising a loop with trip count `E` among `num_threads` threads: we can either give each thread a block of contiguous iterations or we can give each thread every $i$th iteration where $i$ is the number of threads.

Loop flattening (also called loop coalescing) is a technique which has been used in a variety of contexts in the past. For example, to parallelise "irregular, recurrent loop nests" ([9]), to reduce scheduling overhead in Guided self-scheduling ([11]) and more recently in [14] which explores loop flattening in 'coarse-grained reconfigurable architectures' (CGRAs) which must contend with many of the same issues as GPUs albeit in a more extreme fashion.

Loop flattening is the opposite transformation to optimisations like code motion and loop tiling, rather than moving code out of the innermost loop so it is executed less frequently or splitting loops to create extra loops in the nest we push all the code into the innermost loop and then flatten the loops. This always involves extra work since if nothing else we end up recalculating the loop indices of the outer loops on every iteration of the flattened loop. However, performing this extra work can be worth it if it allows us to parallelise the loop.

In fact our case is almost the perfect situation for loop flattening, the iteration space loops are always fully parallel, have known bounds and do not contain other code or conditionals

[2]PyOP2 supports an arbitrary number of 'iteration space' loops in which the $j$ and $k$ loops appear in the benchmarks we consider, our implementation of loop flattening also supports multiple iteration space loops, although this is not evaluated in the context of this work.

between the iteration space loops[3]. The only downside is the possibly expensive integer devide and modulo operations that loop flattening requires.

Finally listings 6.14 and 6.15 show how we combine these techniques to generate the kernel wrapper codes for $\{\text{mass}, \text{quadrilateral}, \text{degree } 4\}$ matching the two alternative strategies we suggested above.

Listing 6.9: Example loop nest

```
1 for (int i=0; i<I; i++) {
2     for (int j=0; j<J; j++) {
3         for (int k=0; k<K; k++) {
4             // Code
5         }
6     }
7 }
```

Listing 6.10: Collapsed loop nest

```
1 for (int e=0; e<I*J*K; e++) {
2     int i = e / (J*K);
3     int j = (e / K) % J;
4     int k = e % K;
5     // Code
6 }
```

Listing 6.11: Chunked Strategy Example

```
1  int tid = threadIdx.x % num_threads;
2  int part_size = ceil(E /
     ↪ num_threads);
3  for (int e=part_size * tid;
4          e<part_size * (tid+1);
5          e++) {
6      if (e > E) {
7          break;
8      }
9      // Code
10 }
```

Listing 6.12: Coalesced Stratagy Example

```
1 int tid = threadIdx.x % num_threads;
2 int part_size = ceil(E /
    ↪ num_threads);
3 for (int e=tid; e<E; e+=part_size) {
4     // Code
5 }
```

### 6.4.1 Number of threads per element

The 'coalesced' scheme described above can allocate an arbitrary number of threads to a tensor however in some situations these threads are predicated off and the work that could be done by them is wasted. For example if the tensor has nine entries and ten threads are allocated to it then the extra thread is always predicated off and the work that could be done by it is wasted. A similar situation (illustrated in figure 6.11) occurs if five threads are allocated the tensor, on the second iteration of the $e$ loop we waste the work of one thread.

The above examples could reduce performance by 10% and 5% percent respectively but the situation could be worse if dividing the number of tensor entries by the number of

[3]Local assembly kernels may contain conditionals but these do not effect the execution of iteration space loops.

Listing 6.13: Critical Section of kernel wrapper for benchmark {mass, quadrilateral, degree 4}

```
for (int idx=threadIdx.x; idx<nelem; idx += blockDim.x) {
  ind_arg1_vec[0] = ind_arg1_shared + loc_map[0*set_size+idx+offset_b]*2 + 0;
  // ...load vertex coordinates into local memory...
  ind_arg1_vec[7] = ind_arg1_shared + loc_map[3*set_size+idx+offset_b]*2 + 1;

  for (int i0=0; i0<25; ++i0 ) {
    for (int i1=0; i1<25; ++i1 ) {
        form_cell_integral_0_otherwise(
          (double (*)[1])(arg0+arg0_lmaoffset+(ele_offset+idx)*625+i0*25+i1*1),
          ind_arg1_vec,
          i0, i1
      );
    }
  }
}
```

Listing 6.14: Kernel Wrapper Chunked Stratagy

```
for (int idx=threadIdx.x/25; idx < nelem; idx += blockDim.x/25) {
  if (threadIdx.x >= 125) { // blockDim.x - blockDim.x % 25
    continue;
  }
  ind_arg1_vec[0] = ind_arg1_shared + loc_map[0*set_size+idx+offset_b]*2 + 0;
  // ...load vertex coordinates into local memory...
  ind_arg1_vec[7] = ind_arg1_shared + loc_map[7*set_size+idx+offset_b]*2 + 1;

  int local_thread_id = threadIdx.x % 25;
  for (int e=25 * local_thread_id; e<25 * (local_thread_id+1); e++) {
      int i0 = e / 25;
      int i1 = e % 25;
      form_cell_integral_0_otherwise(
        (double (*)[1])(arg0+arg0_lmaoffset+(ele_offset+idx)*625+i0*25+i1*1),
        ind_arg1_vec,
        i0, i1
      );
  }
}
```

Listing 6.15: Kernel Wrapper Coalesced Stratagy

```
for (int e=25 * local_thread_id; e<25 * (local_thread_id+25); e++) {
  if (threadIdx.x >= 125) { // 125 == blockDim.x - blockDim.x % 25
    continue;
  }
  ind_arg1_vec[0] = ind_arg1_shared + loc_map[0*set_size+idx+offset_b]*2 + 0;
  // ...load vertex coordinates into local memory...
  ind_arg1_vec[7] = ind_arg1_shared + loc_map[7*set_size+idx+offset_b]*2 + 1;

  int local_thread_id = threadIdx.x % 25;
  for (int e=local_thread_id; e<625; e+=25) {
      int i0 = e / 25;
      int i1 = e % 25;
      form_cell_integral_0_otherwise(
        (double (*)[1])(arg0+arg0_lmaoffset+(ele_offset+idx)*625+i0*25+i1*1),
        ind_arg1_vec,
        i0, i1
      );
  }
}
```

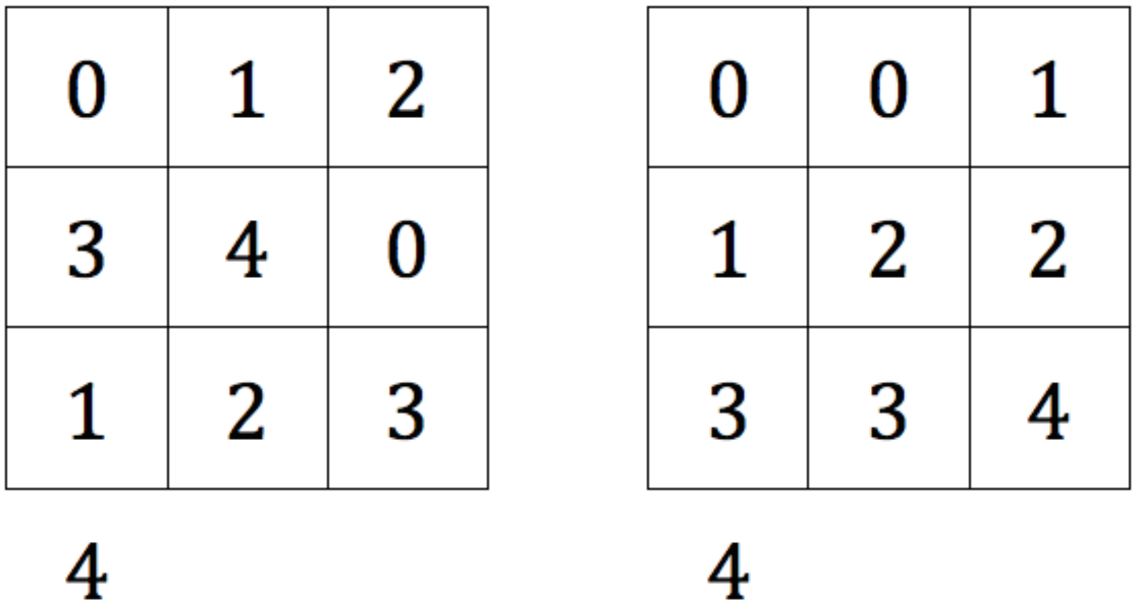



Figure 6.11: Allocating five threads to a nine entry tensor under the 'coalesced' and 'chunked' strategies

threads leaves a large remainder. We can not waste the work of more than 31 threads this way since 32 contiguous threads together form a warp and a whole warp can not be predicated off, it simply does not execute the instructions.

A similar situation occurs for the 'chunked' strategy however in this case we waste one thread for $x$ iterations of the $e$ loop rather than $x$ threads for one iteration (see figure 6.11).

Finally the same problem occurs with the number of threads per element and the block size: if the number of threads per element is not a divisor of the block size then some threads at the 'end' of the block must be predicated off permanently since otherwise they would compute some but not all of the entries in a local tensor while other threads are trying to compute and non-atomically write the same locations.

These two facts together suggest that we could only get full performance when the number of threads per element is a divisor of both the block size and of the tensor size. We can build a model to quantify the slowdown we expect when this is not the case based on the factors mentioned above.

These problems could be avoided entirely by not assigning threads to specific elements. Instead we could have the 128 threads in a block compute the first 128 tensor entries, then compute the next 128 tensor entries etc, irrespective of which tensors they came from. Although this avoids the problem it also obscures one of the most interesting possibilities of multiple threads per element, which is having threads share data while computing the tensor. For example the expression we evaluate in the innermost loop can contain subexpressions common across entries in the local tensor, we could extract these, have

different threads compute different common subexpressions then share these via shared memory or warp shuffle instructions. This becomes significantly more difficult if threads do not know which of their neighbours are working on the same element. We expand on this topic in section 7.3.

### Hypothesis

When the number of threads allocated to an element is not a divisor of the number of entries in the local tensor and the block size performance should decrease proportional to the amount of work wasted.

### Experiment

We benchmark the coalesced and the chunked scheme described at the start of section 6.4 with 1 to 128 threads per element and compare the results to a theoretical model of the amount of work wasted.

### Discussion

The experiments show that the model accurately predicts slowdowns, when a significant slowdown is predicted there really is a significant slowdown in the actual results, but that the model is also over optimistic - frequently predicting no slowdown where there are significant slowdowns.

Two types of discontinuities are visible as vertical lines both in the graphs showing the experimental results, figures 6.12 and 6.13 and in figure 6.14 which shows the predicted results. These lines mirror the two factors discussed above. The contiguous lines which run from the top to the bottom of the graphs are due to the interaction between the block size and number of threads per element, this is why these lines appear in the same place across every benchmark, they are not effected by the changing tensor size, and why they appear on the $64-65$, $42-43$ and $32-33$ (among other) boundaries - these are all places where crossing the boundary makes a significant difference to the quotient of the block size divided by the number of threads per element and hence to the performance. The discontinuities shared by benchmarks with the same tensor size are due to the interaction between the tensor size and the number of threads per element and occur for the same reasons.

We conclude that avoiding poor performance under either the coalesced or chunked scheme requires at least avoiding a number of threads per element which interacts poorly with the block size or the tensor size but that is a necessary condition not a sufficient one.

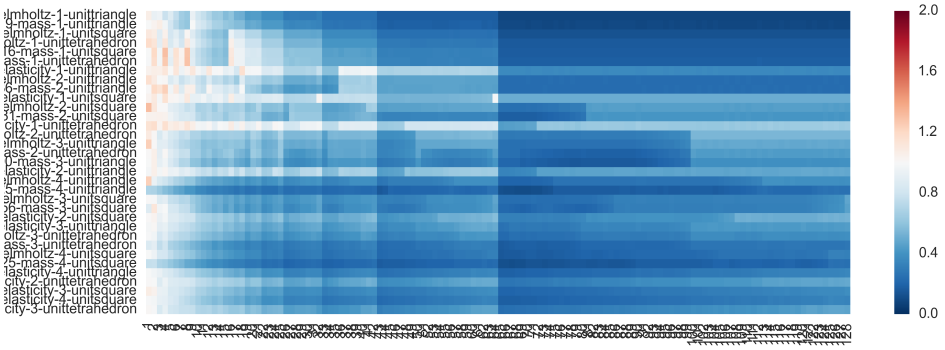



Figure 6.12: Speedup for each benchmark under the 'coalesced' scheme (compared to post-Constant Hoisting performance) for 1 to 128 threads

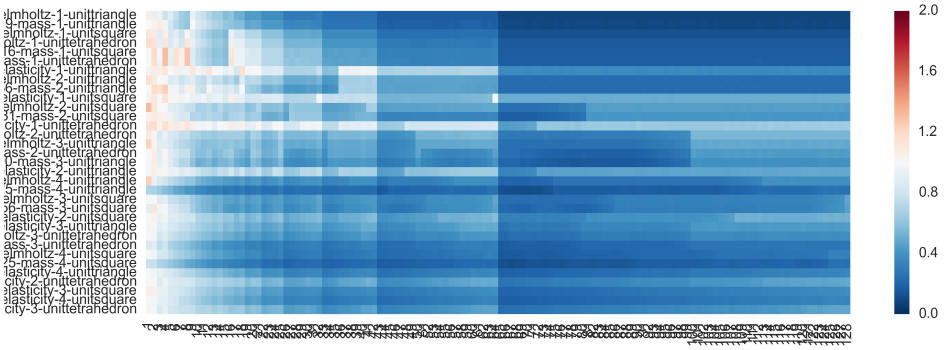



Figure 6.13: Speedup for each benchmark under the 'chunked' scheme (compared to post-Constant Hoisting performance) for 1 to 128 threads

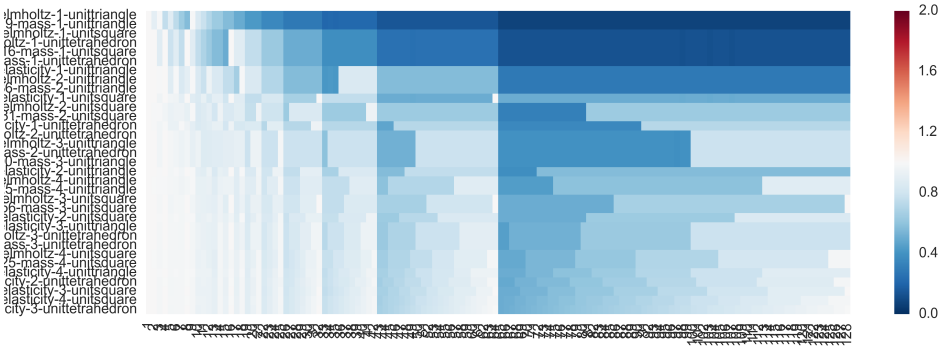



Figure 6.14: Theoretical speedup for each benchmark for 1 to 128 threads

### 6.4.2 Special Case Loop Flattening

Our investigation in sub-section 6.4.1 suggests we should choose a number of threads per element which is a divisor of the tensor size. The tensor size itself and the trip counts of the $j$ or $k$ loops[4] are obvious divisors of tensor size and in these cases the generated code could be simplified significantly.

In the case where the number of threads assigned to an element is equal to the tensor size (and so the total number of iterations of the flattened loop) we can remove the loop entirely, see listing 6.17.

Listing 6.16: With redundant loop

```
1 int tid = threadIdx.x % 9;
2 for (int e=tid; e<9; e+=9) {
3     int i = e / 3;
4     int j = e % 3;
5     // Code
6 }
```

Listing 6.17: Without redundant loop

```
1 int local_thread_id = threadIdx.x %
    ↪ 9;
2 int i = e / 3;
3 int j = e % 3;
4 // Code
5 }
```

More generally if the number of threads per element is equal to the product of some subset of the trip counts of the iteration space loops we can replace those loops with code to calculate the indices directly but leave the rest of the loops. This preserves a much more natural formulation than the flattened loop (see listing 6.19).

With either of these two simplifications we can choose to preserve the coalesced or the chunked access patten. Removing loops from innermost to outermost (as in listing 6.19) gives the coalesced access pattern while removing loops from outermost to innermost gives the chunked access pattern.

Listing 6.18: Flattened Loop Formulation

```
1 int tid = threadIdx.x % 3;
2 for (int e=tid; e<9; e+=3) {
3     int i = e / 3;
4     int j = e % 3;
5     // Code
6 }
```

Listing 6.19: Natural Loop Formulation

```
1 for (int i=0; i<3; i++) {
2     int j = threadIdx.x % 3;
3     // Code
4 }
```

#### Hypothesis

We hypothesis that these simplifications make no difference to the performance after all they only modify some of the book keeping around the computation and do not change the order or how the entires of the local tensor are computed.

[4] And more generally in PyOP2 the trip count of any iteration space loop.

#### Experiment

We implemented the two simplifications described above and, for each benchmark where these simplifications are relevant, compare performance with and without the simplification.

#### Discussion

Unfortunately the results (6.4.2) show that these simplifications can significantly out perform (up to almost six times) the non-special cased versions. The effect is perticually bad for the mass problem but also effects the other two problems. This suggests that our loop flattening technique carries significant overheads either directly, via the additional mod and divide instructions or indirectly via `nvcc` generating worse code for non standard loop constructs.

To conduct the remainder of our investigation into multiple threads per element we had to choose a method for assigning a number of threads to an element. Based on subsection 6.4.1 we initially chose the greatest common divisor of the tensor size and the block size however in many situations they had no common divisor greater than one. Instead we use a strategy 'gcd10' which chooses the maximum of the greatest common divisor of: `gcd(tensorsize, blocksize)`, `gcd(tensorsize, blocksize-1)`, ... `gcd(tensorsize, blocksize-9)`. This allows a significant number of threads to be assigned to each benchmark and by our model causes a maximum performance drop of 10% and normally a much smaller drop. This is almost certainly not the optimum choice but it allowed us to continue our investigation into the effects of using many threads to perform assembly.

### 6.4.3 Chunked vs. Coalesced

The two alternative strategies we have considered are two different thread orderings for computing the elements of the local matrix. These two orderings correspond to two different patterns for writing out the data to the underlying array. The coalesced scheme writes out the data in a sequential way (in the sense that threads 1, 2 and 3 write to locations 1, 2 and 3 in the array) the chunked scheme writes out the data in strided way (while thread 1 is writing location 1 thread 2 is writing location 3 and thread 3 is writing location 5 etc. Figure 6.15 illustrates the difference.

In order to access global memory efficiently warps must access memory in a way that allows coalescing falling to do this causes the accesses to be serialised hurting performance.

The 'coalesced' pattern (unsurprisingly) accesses the underlying array in the suggested way for coalescing. The chunked pattern is regular but strided. In general the amount of coalescing possible for a strided access pattern decreases as the length of the stride increases.

| Benchmark | Meshtype | Degree | Tensor Size | Chunked Column | Matrix | Coalesced Row | Matrix |
|---|---|---|---|---|---|---|---|
| mass | Triangle | 1 | 9 | 1.17 | 1.2 | 1.18 | 1.0 |
| mass | Triangle | 2 | 36 | 1.73 | 2.06 | 1.77 | 1.0 |
| mass | Triangle | 3 | 100 | 2.39 | 2.72 | 2.93 | 1.0 |
| mass | Triangle | 4 | 225 | 3.68 | | 5.28 | |
| mass | Square | 1 | 16 | 1.25 | 1.46 | 1.22 | 1.0 |
| mass | Square | 2 | 81 | 1.76 | 2.17 | 2.18 | 1.0 |
| mass | Square | 3 | 256 | 2.97 | | 3.87 | |
| mass | Square | 4 | 625 | 4.69 | | 5.79 | |
| mass | Tetrahedron | 1 | 16 | 1.2 | 1.32 | 1.21 | 1.11 |
| mass | Tetrahedron | 2 | 100 | 2.01 | 1.9 | 2.07 | 1.32 |
| mass | Tetrahedron | 3 | 400 | 3.24 | | 3.81 | |
| mass | Tetrahedron | 4 | 1225 | 3.15 | | 3.26 | |
| helmholtz | Triangle | 1 | 9 | 1.16 | 1.22 | 1.16 | 1.09 |
| helmholtz | Triangle | 2 | 36 | 1.38 | 1.66 | 1.44 | 1.24 |
| helmholtz | Triangle | 3 | 100 | 1.05 | 0.99 | 1.04 | 1.0 |
| helmholtz | Triangle | 4 | 225 | 1.02 | | 1.02 | |
| helmholtz | Square | 1 | 16 | 1.18 | 1.32 | 1.19 | 1.14 |
| helmholtz | Square | 2 | 81 | 1.55 | 1.56 | 1.58 | 1.21 |
| helmholtz | Square | 3 | 256 | 1.02 | | 1.01 | |
| helmholtz | Square | 4 | 625 | 1.01 | | 1.03 | |
| helmholtz | Tetrahedron | 1 | 16 | 1.06 | 1.06 | 1.05 | 1.03 |
| helmholtz | Tetrahedron | 2 | 100 | 1.03 | 0.98 | 1.03 | 0.99 |
| helmholtz | Tetrahedron | 3 | 400 | 1.01 | | 1.01 | |
| elasticity | Triangle | 1 | 36 | 1.14 | 1.27 | 1.14 | 1.12 |
| elasticity | Triangle | 2 | 144 | 1.63 | | 1.66 | |
| elasticity | Triangle | 3 | 400 | 1.02 | | 1.03 | |
| elasticity | Triangle | 4 | 900 | 1.0 | | 1.0 | |
| elasticity | Square | 1 | 64 | 1.17 | 1.2 | 1.15 | 1.09 |
| elasticity | Square | 2 | 324 | 1.04 | | 1.05 | |
| elasticity | Square | 3 | 1024 | 1.0 | | 1.0 | |
| elasticity | Square | 4 | 2500 | 0.99 | | 1.0 | |
| elasticity | Tetrahedron | 1 | 91 | 1.15 | | 1.16 | |
| elasticity | Tetrahedron | 2 | 1006 | 1.0 | | 1.01 | |
| elasticity | Tetrahedron | 3 | 6752 | 0.95 | | 0.94 | |

Table 6.7: Speedups for Special Case Loop Flattening vs. Loop Flattening

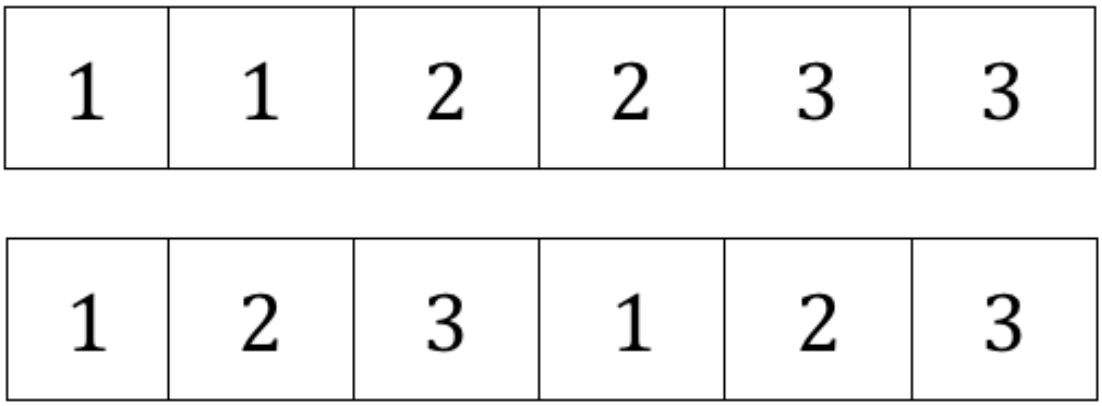



Figure 6.15: Illustration of the how the chunked (top) and coalesced (bottom) schemes access a six entry local matrix with three threads.

### Hypothesis

The coalesced scheme should perform better than the chunked scheme due to better global store coalescing.

### Experiment

We can benchmark the two schemes and compare them. Then profile an example benchmark to examine the reason for the performance difference if any.

### Discussion

We can see from figure 6.16 that the coalesced scheme universally has the same or better performance than the chunked scheme, in some cases 1.8 times. Table 6.8 shows the 'efficiency' of the memory accesses[5] under the normal one thread per element scheme, the chunked scheme and the coalesced scheme for the {mass,triangle,4, o, degree $n$} a 10,000 element mesh.

| | **Shared Memory Efficiency** | **Global Load Efficiency** | **Global Store Efficiency** |
|---|---|---|---|
| **Baseline** | 30.0% | 25.2% | 25.0% |
| **Chunked** | 97.6% | 29.4% | 25.0% |
| **Coalesced** | 97.6% | 100.0% | 91.2% |

Table 6.8: Memory Access Efficiency for {mass,triangle,4, o, degree $n$} a 10,000 element mesh

Both the coalesced and the chunked scheme are much more efficient in their use of shared memory. This makes sense, the only use of shared memory in the kernel is reading the

[5]Efficiency is a metric that compares the number of required memory transactions to the theoretical minimum if they were perfectly coalesced for some kernel execution.

staged the vertex coordinates and previously every thread accessed different vertex coordinates (hence the poor shared memory efficiency of the baseline implementation) now most threads in the block access the same vertex coordinates since they are working on the same element (accesses where every thread in a warp reads the same address are perfectly coalesced). This number does not tell the whole story we now make $n$ times as many accesses to the shared memory as we did before where $n$ is the number of threads assigned to an element.

As we expected the coalesced scheme has much better global read and write efficiency while the chunked scheme's efficiency is closer to the baseline implementation.

These results show that it is both possible and important for performance to ensure that the scheme for assigning multiple threads to an element results in coalesced memory accesses.

### 6.4.4 Parameter Tuning and Multiple Threads per Element

Our investigation into using multiple threads per element has shown how we can expose the additional parallelism as efficiently as possible. However as we can see in figure 6.17 and figure 6.18 that alone this only helps for low polynomial elements. This is as we might expect, since we always want to consider the largest meshes possible there is already plenty of parallelism however there are advantages to this new source of parallelism it is 'cheaper' then the per element kind requiring fewer resources per thread. This suggests that we should now be able to fit more blocks into each SM allowing us to take better advantage of the hardware.

#### Hypothesis

Using multiple threads per element increases the amount of utilizable parallelism and choosing the correct set of parameters allows the GPU to take the best advantage of the available parallelism so these optimisations ought to be mutually beneficial and combining them should increase performance.

#### Experiment

We performed a parameter sweep as in section 6.2 using the 'coalesced gcd10' scheme for multiple threads per element described in section 6.4.

#### Discussion

Figure 6.20 shows the best achieved speedup after parameter tuning compared to the baseline. Figure 6.19 shows the average speedup for each set of parameters, comparing this

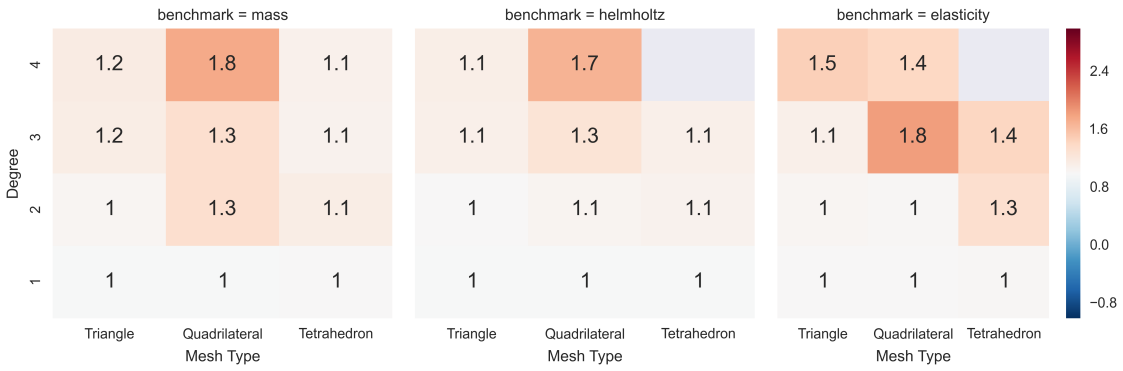



Figure 6.16: The speedup for the 'coalesced gcd10' scheme compared to the 'chunked gcd10' scheme across all benchmarks.

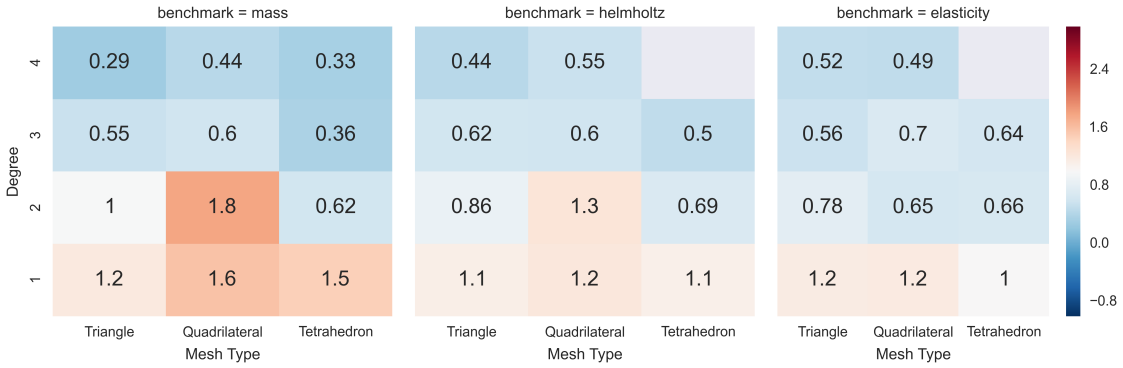



Figure 6.17: The speedup for the 'coalesced gcd10' scheme compared to the post-constant hoisting baseline.

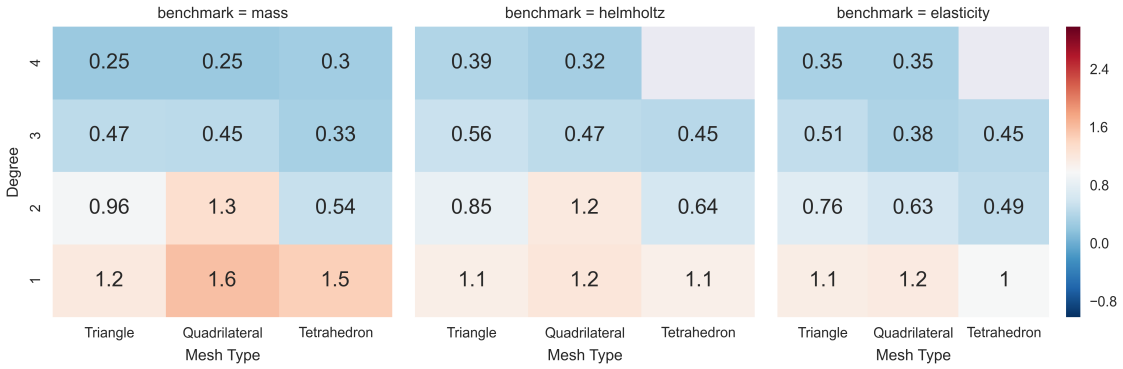



Figure 6.18: The speedup for the 'chunked gcd10' scheme compared to the scheme compared to the post-constant hoisting baseline.

with figure 6.4 we see that using multiple threads per element has significantly increased the range of acceptable parameters which give reasonable performance improvements, we can now use much larger block sizes while still increasing the partition fraction.

The most interesting graph is figure 6.21 which compares the performance of the gcd10 scheme with parameter tuning to the best results from parameter tuning alone. It shows that using multiple threads per element can give up to two times improvement over just parameter tuning and that in general it improves the performance of the low order benchmarks and mass the benchmarks significantly. These are the benchmarks which have the lowest arithmetic intensity suggesting this optimisation is most helpful in cases where the problems have lower arithmetic intensity.

Ideally we would repeat the previous parts of this investigation parameter tuning at each opportunity however this will have to wait for future work.

This concludes our investigation of the four optimisations.

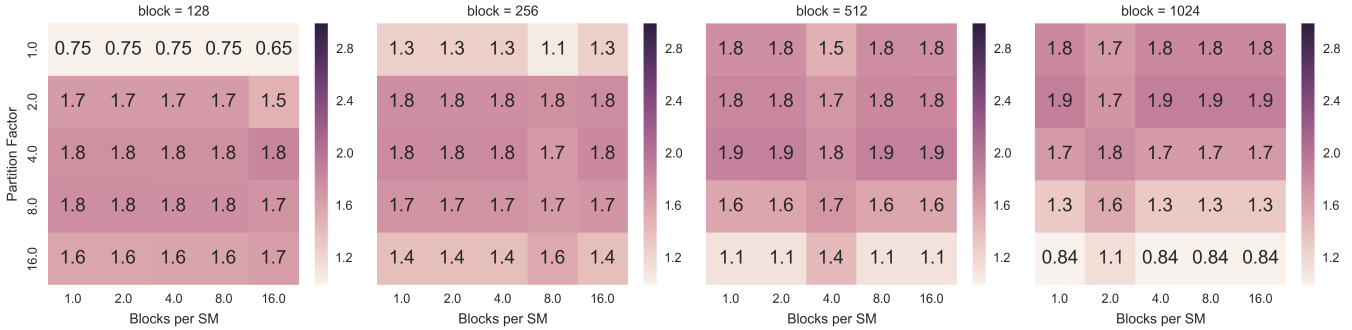



Figure 6.19: The average speedup of the 'coalesced gcd10' scheme (compared to the post-Constant Hoisting baseline) across all benchmarks for each parameter set

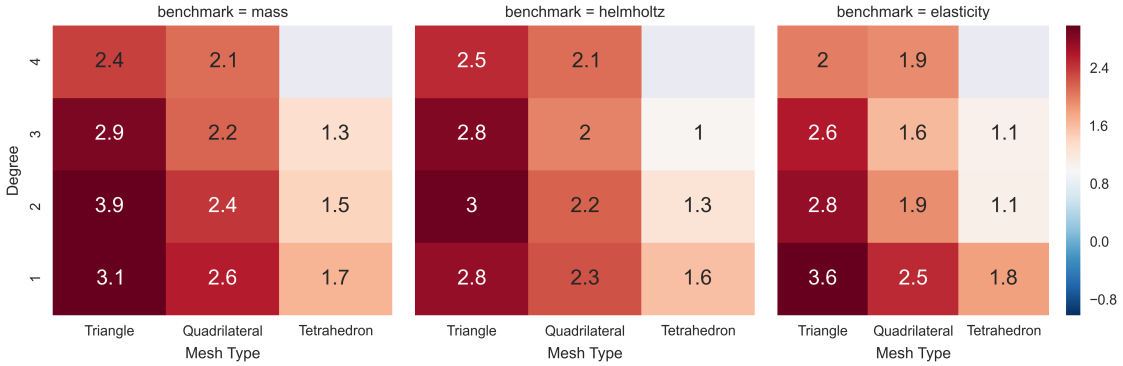



Figure 6.20: Maximum speedup of the 'coalesced gcd10' scheme compared to the post-constant hoisting baseline for each benchmark after a parameter sweep

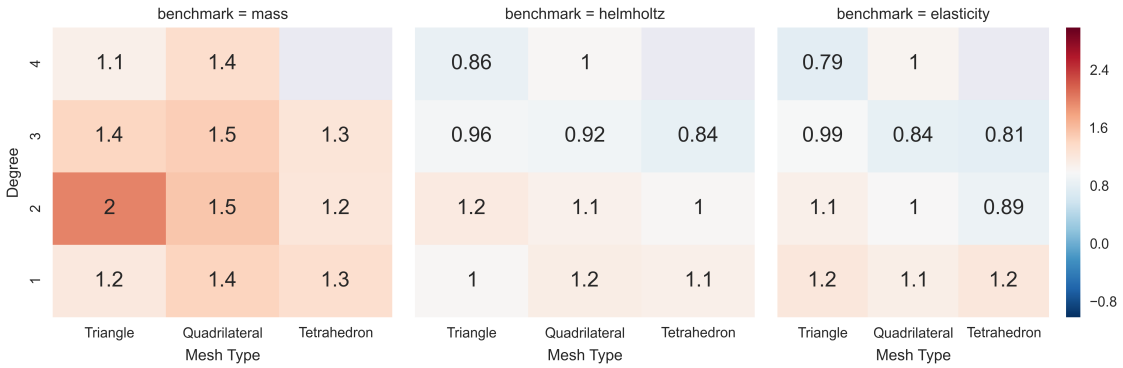



Figure 6.21: Maximum speedup of the 'coalesced gcd10' scheme after a parameter sweep compared to the best post-constant hoisting baseline after a parameter sweep

# 7 Conclusion

Our evaluation of the four individual optimisations has been conducted inline with our investigation. We can now consider the best possible combination of these optimisations and evaluate the project as a whole.

Table 7.1 shows for each benchmark the best combination of the optimisations we considered, this speedup is visualised in figure 7.1. To see what performance we have achieved in absolute terms we used `nvprof` to measure the number of double precision floating point operations the kernels and then computed the FLOPs for each benchmark, this is shown in figure 7.2. The results are broadly as we would expect more complex problems with high order basis function and larger elements have greater operational intensity and so achieve higher FLOPs.

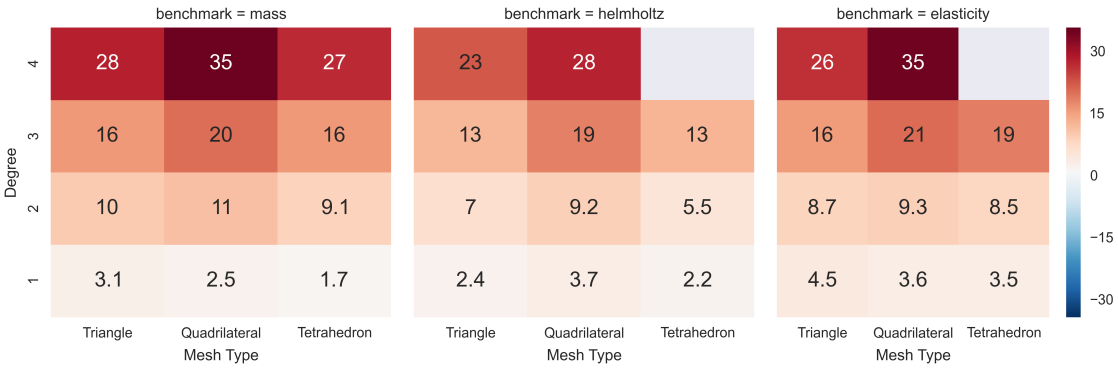



Figure 7.1: Best achieved speedup compared to Firedrake's current performance for each benchmark

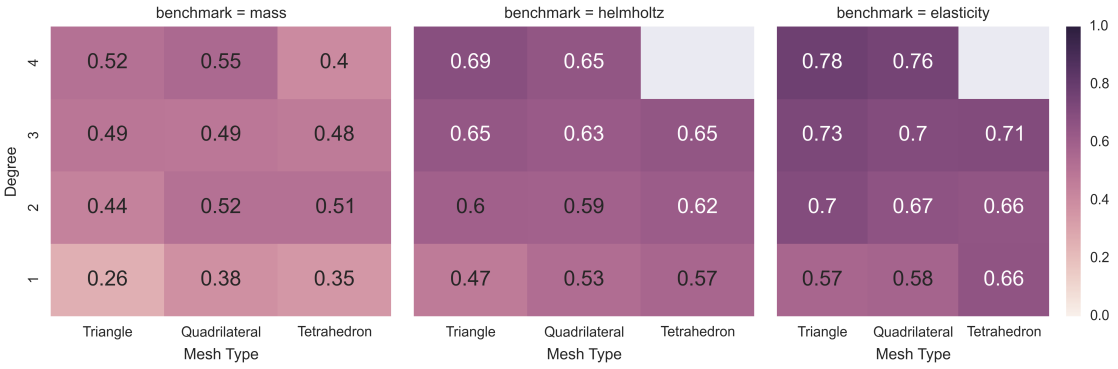



Figure 7.2: FLOPs for best combination of the optimisations for each benchmark

## 7.1 Contributions

To reiterate the main contributions of this thesis are as follows:

- We show that storing the basis functions in local memory is catastrophic to the performance of problems with $>$ 1st degree basis functions, and that using constant hoisting to move the basis functions into constant memory can produce speedups of up to eighteen times. We produce patches allowing this optimisation to be immediately incorporated into Firedrake and PyOP2.
- We investigate parameter tuning to improve the occupancy of the generated CUDA kernels. We find that PyOP2's current strategy for choosing the three critical parameters which effect occupancy: registers per thread, blocksize and elements per block is not optimal for any of the benchmarks studied. We show that better choice of these parameters can result in a threefold performance improvement in some cases and that while the best results (1.93 times average speedup) are only achieved by tuning the parameters individually for each problem we can choose a single set of parameters to achieve an average speedup of 1.80 times and improve the performance of all but one benchmark.
- We examine the effect of forbidding and encouraging loop unrolling using `pragma unroll`. We discover that while loop unrolling is critical to the performance of kernels with low arithmetic intensity only in a minority of cases where the loop trip counts are large enough that CUDA is reluctant to unroll them by default does encouraging CUDA to unroll them improve performance and this improvement in of the order of 10 to 30%. We also show that CUDA frequently ignores the pragma especially for nested loops with large trip counts.
- We present a novel technique for allowing any number of threads up to one a limit of one per local matrix entry to corporate on assembling an element. The technique flattens the $j$ and $k$ loops and assigns sections of the flattened loop to each thread. We show that for this technique to be effective the number of threads per element must be a must be a divisor of the tensor size and close to a divisor of the block size and we show the necessity of coalesced memory accesses in this scheme. We show that combining this approach with parameter tuning improves the performance of low order and low arithmetic intensity kernels.
- Finally we consider the optimisations together and show average improvements of 13 times rising to 35 times for some benchmarks. We also show we can achieve an average 57% of peak FLOPs on the NVIDA GRID K520.
- Each of the above results is validated through testing with three separate problems of greatly varying complexity, three element types including 2D and 3D elements, polynomial basis functions of degree 1-4 and the combinations thereof. This provides evidence that these results will generalize to yet unseen FEM problems: one of the key benefits of Firedrake.

## 7.2 Conclusion

Both FEM and Firedrake many deep abstraction which have made this project challenging however we have managed to speed up Firedrake's local assembly on GPUs significantly, up to 35 times for some benchmarks and most of the lessons we have learnt should be broadly applicable for GPU Finite Element assembly in other situations.

We have investigated four optimisations, found one that Firedrake should adopt immediately (constant hoisting) and one (parameter tuning) that should implemented as soon as a good strategy can be found for doing the tuning or a good cost model is proposed (although simply assigning half as many elements to each block would help significantly as a stop gap measure).

We proposed a new strategy for using multiple threads to assemble a single element more work and begun to investigate and although more could done we have shown that for some low arithmetic intensity problems it can give a two times performance improvement.

We also considered (loop unrolling) which is often useful but also often done automatically by `nvcc`. Using `pragma unroll` seemed to be helpful only in limited although circumstances.

Finally we performed a very limited analysis of our optimisations in absolute terms finding we achieved a high percentage of peak FLOPs however one of the weaknesses of this work is that this analysis is not more robust.

## 7.3 Future Work

This work only begins the project of automatically producing optimised local assembly codes for GPUs. We could not consider everything and this work has also opened interesting avenues for future research.

**OpenCL**

One obvious extension to this work is to investigate an OpenCL implementation. In this work we considered CUDA only so we could take advantage of its tooling assuming that the results would apply to OpenCL on the same target architectures given its similar programing model however this assumption requires testing. An OpenCL implementation would also allow us to compare the performance of AMD and NVIDA GPUs.

**COFFEE**

One of the goals of this project which we did not meet was to apply COFFEE's optimisations in the GPU context. Many of COFFEE's optimisations (for example generalized invariant code motion) involve trading increased temporary storage for less computation however GPU have a relative abundance of computation compared to the number of registers per thread so it may be that we want to do the exact opposite of this and trade duplicated computation for decreased register pressure.

COFFEE's other optimisations involve applying vectorisation to the kernel and having explored how to assign multiple threads to an element we now in an excellent position to investigate applying these optimisations for GPUs.

**Function Spaces**

In this work we did not consider additional function spaces which manifest themselves as additional indirect arguments to the kernel. Given that these increase the memory requirements of the kernel we might expect them to benefit more than the benchmarks we considered from the optimisations which assign multiple threads to an element reducing the memory footprint however they would also effect the amount of bandwidth required by the kernels.

**Jacobian**

As well as parallelism in the *element*, $j$ and $k$ loops there is also parallelism present in the computation of the Jacobian. However it is of a different sort to that *element*, $j$ and $k$ loops. It would be interesting consider approaches to take advantage of this. More generally our current approach to the Jacobian, recomputing it for each local matrix element, is clearly suboptimal and some better approach must be found.

| Problem | Meshtype | Degree | Strategy | Blocksize | Partition Factor | SMs per Block | Speedup | Percent of Peak FLOPs |
|---|---|---|---|---|---|---|---|---|
| mass | unittriangle | 1 | gcd10 | 256 | 2 | 1 | 3.101984 | 0.259575 |
| mass | unittriangle | 2 | gcd10 | 128 | 4 | 2 | 10.032070 | 0.440941 |
| mass | unittriangle | 3 | gcd10 | 512 | 4 | 1 | 16.226553 | 0.494637 |
| mass | unittriangle | 4 | gcd10 | 1024 | 2 | 1 | 28.219022 | 0.517076 |
| mass | unitsquare | 1 | gcd10 | 128 | 2 | 1 | 2.538906 | 0.380207 |
| mass | unitsquare | 2 | gcd10 | 128 | 2 | 4 | 11.447474 | 0.516954 |
| mass | unitsquare | 3 | gcd10 | 256 | 8 | 2 | 20.438051 | 0.486861 |
| mass | unitsquare | 4 | gcd10 | 512 | 4 | 4 | 35.299448 | 0.551039 |
| mass | unittetrahedron | 1 | gcd10 | 256 | 1 | 1 | 1.651056 | 0.349711 |
| mass | unittetrahedron | 2 | gcd10 | 512 | 2 | 1 | 9.057831 | 0.514752 |
| mass | unittetrahedron | 3 | gcd10 | 1024 | 1 | 1 | 16.054782 | 0.475285 |
| mass | unittetrahedron | 4 | base | 256 | 1 | 1 | 27.083580 | 0.404242 |
| helmholtz | unittriangle | 1 | gcd10 | 128 | 2 | 4 | 2.387420 | 0.472191 |
| helmholtz | unittriangle | 2 | gcd10 | 128 | 4 | 16 | 7.012997 | 0.601522 |
| helmholtz | unittriangle | 3 | base | 256 | 2 | 8 | 12.633113 | 0.646815 |
| helmholtz | unittriangle | 4 | base | 256 | 2 | 8 | 22.801415 | 0.688819 |
| helmholtz | unitsquare | 1 | gcd10 | 256 | 2 | 16 | 3.669533 | 0.534388 |
| helmholtz | unitsquare | 2 | gcd10 | 256 | 2 | 1 | 9.229746 | 0.592307 |
| helmholtz | unitsquare | 3 | base | 128 | 4 | 16 | 18.564293 | 0.626643 |
| helmholtz | unitsquare | 4 | gcd10 | 512 | 4 | 4 | 27.998778 | 0.650470 |
| helmholtz | unittetrahedron | 1 | gcd10 | 256 | 2 | 1 | 2.214882 | 0.568903 |
| helmholtz | unittetrahedron | 2 | gcd10 | 128 | 4 | 1 | 5.466639 | 0.618567 |
| helmholtz | unittetrahedron | 3 | base | 256 | 1 | 1 | 12.850643 | 0.654448 |
| elasticity | unittriangle | 1 | gcd10 | 512 | 4 | 2 | 4.450640 | 0.569368 |
| elasticity | unittriangle | 2 | gcd10 | 1024 | 2 | 16 | 8.671818 | 0.703238 |
| elasticity | unittriangle | 3 | base | 128 | 4 | 16 | 15.870925 | 0.733239 |
| elasticity | unittriangle | 4 | base | 128 | 4 | 16 | 26.457563 | 0.780266 |
| elasticity | unitsquare | 1 | gcd10 | 256 | 2 | 16 | 3.573897 | 0.579661 |
| elasticity | unitsquare | 2 | gcd10 | 128 | 4 | 16 | 9.344588 | 0.665119 |
| elasticity | unitsquare | 3 | base | 128 | 4 | 16 | 20.835831 | 0.704890 |
| elasticity | unitsquare | 4 | gcd10 | 512 | 8 | 4 | 34.940822 | 0.757369 |
| elasticity | unittetrahedron | 1 | gcd10 | 1024 | 1 | 16 | 3.520061 | 0.663072 |
| elasticity | unittetrahedron | 2 | base | 256 | 1 | 2 | 8.491247 | 0.660029 |
| elasticity | unittetrahedron | 3 | base | 256 | 1 | 4 | 18.787962 | 0.712535 |

Table 7.1: Best combined optimisation for each benchmark

# Glossary

**AST** Abstract Syntax Tree.

**COFFEE** An optimising compiler for local assembly kernels. See section 2.3.

**CPU** Central Processing Unit.

**DSL** Domain Specific Language.

**FEM** Finite Element Methods.

**FFC** FEniCS Form Compiler. Compiles UFL to C, see section 2.2.2.

**Firedrake** An automated system for solving partial differential equations. See section 2.2.

**GPU** Graphical Processing Unit.

**NVidia** An American company that produces GPUs.

**PDE** Partial Differential Equation.

**PETSc** See section 2.2.4.

**PyOP2** See section 2.2.3.

**SIMD** Single Instruction Multiple Data.

**SIMT** Single Instruction Multiple Threads.

**SM** Streaming Multiprocessors.

**SMT** Simultaneous Multithreading.

**UFL** Unified Form Language.

# Bibliography


[1] Martin S Alnæs et al. "Unified Form Language: A Domain-specific Language for Weak Formulations of Partial Differential Equations". In: *ACM Trans. Math. Softw.* 40.2 (Mar. 2014), 9:1–9:37. ISSN: 0098-3500. DOI: 10.1145/2566630. arXiv: 1112.0402. URL: http://arxiv.org/abs/1112.0402.

[2] Krzysztof Bana, Przemyslaw Plaszewski, and Pawel Maciol. "Numerical integration on GPUs for higher order finite elements". In: *Pre-Print* (2013). arXiv: arXiv:1310.1191v1.

[3] Cris Cecka, Adrian J. Lew, and Eric Darve. "Assembly of finite element methods on graphics processors". In: *International Journal for Numerical Methods in Engineering* (2010), n/a–n/a. ISSN: 00295981. DOI: 10.1002/nme.2989. URL: http://doi.wiley.com/10.1002/nme.2989.

[4] Cris Cecka, Adrian Lew, and Eric Darve. "Application of assembly of finite element methods on graphics processors for real-time elastodynamics". In: (2011).

[5] Advanced Micro Devices. *Porting CUDA Applications to OpenCL*. 2014. URL: http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/programming-in-opencl/porting-cuda-applications-to-opencl (visited on 02/03/2014).

[6] G. Dhatt, E. Lefrançois, and G. Touzot. *Finite Element Method*. ISTE. Wiley, 2012. ISBN: 9781118569702. URL: http://books.google.co.uk/books?id=wvE1ClcHq5IC.

[7] Adam Dziekonski et al. "Finite element matrix generation on a GPU". In: *Progress In Electromagnetics Research* 128 (2012), pp. 249–265.

[8] J Filipovic, I Peterlík, and Jan Fousek. "GPU Acceleration of equations assembly in finite elements method-preliminary results". In: *SAAHPC: Symposium on Application ...* (2009). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.157.5693%5C&rep=rep1%5C&type=pdf.

[9] Anwar M. Ghuloum and Allan L. Fisher. "Flattening and Parallelizing Irregular, Recurrent Loop Nests". In: *SIGPLAN Not.* 30.8 (Aug. 1995), pp. 58–67. ISSN: 0362-1340. DOI: 10.1145/209937.209944. URL: http://doi.acm.org/10.1145/209937.209944.

[10] G. Karniadakis and S. Sherwin. *Spectral/hp element methods for computational fluid dynamics*. 2nd ed. Oxford University Press, 2005, pp. 1–650. ISBN: 9780198528692.

[11] Arun Kejariwal, Alexandru Nicolau, and ConstantineD. Polychronopoulos. "Enhanced Loop Coalescing: A Compiler Technique for Transforming Non-uniform Iteration Spaces". English. In: *High-Performance Computing*. Ed. by Jesús Labarta, Kazuki Joe, and Toshinori Sato. Vol. 4759. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 17–32. ISBN: 978-3-540-77703-8. DOI: 10.1007/978-3-540-77704-5_2. URL: http://dx.doi.org/10.1007/978-3-540-77704-5_2.

[12] Dimitri Komatitsch et al. "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster". In: *Journal of computational physics* 229.20 (2010), pp. 7692–7714.

[13] Dimitri Komatitsch et al. "Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs". In: *Computer Science-Research and Development* 25.1-2 (2010), pp. 75–82.

[14] Jongeun Lee et al. "Flattening-based mapping of imperfect loop nests for CGRAs?" In: *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2014 International Conference on*. Oct. 2014, pp. 1–10. DOI: 10.1145/2656075.2656085.

[15] Anders Logg. "Automating the finite element method". In: *Archives of Computational Methods in Engineering* 14.2 (2007), pp. 93–138.

[16] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*. Vol. 84. Springer Science & Business Media, 2012.

[17] Fabio Luporini et al. "COFFEE: an Optimizing Compiler for Finite Element Local Assembly". In: *Submitted to ACM Transactions on Architecture and Code Optimization* (2014). URL: http://arxiv.org/abs/1407.0904.

[18] Fabio Luporini et al. "Cross-Loop Optimization of Arithmetic Intensity for Finite Element". In: *Mathematical Software* 11.4 (2015), p. 57. arXiv: 1407.0904.

[19] Paweł Macioł, Przemysław Płaszewski, and Krzysztof Banaś. "3D finite element numerical integration on GPUs". In: *Procedia Computer Science* 1.1 (May 2010), pp. 1093–1100. ISSN: 18770509. DOI: 10.1016/j.procs.2010.04.121. URL: http://linkinghub.elsevier.com/retrieve/pii/S1877050910001225.

[20] G. R. Markall et al. "Finite element assembly strategies on multi- and many-core architectures". In: *International Journal for Numerical Methods in Fluids* 71 (2013), pp. 80–97. DOI: 10.1002/fld.3648. URL: http://dx.doi.org/10.1002/fld.3648.

[21] Graham R. Markall et al. "Performance-Portable Finite Element Assembly Using PyOP2 and FEniCS". In: *28th International Supercomputing Conference, ISC, Proceedings*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Vol. 7905. Lecture Notes in Computer Science. Springer, 2013, pp. 279–289. DOI: 10.1007/978-3-642-38750-0_21. URL: http://dx.doi.org/10.1007/978-3-642-38750-0_21.

[22] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. 2014.

[23] Florian Rathgeber et al. "Firedrake: automating the finite element method by composing abstractions". In: *Submitted to ACM TOMS* (2015). arXiv: 1501.01809.

[24] Florian Rathgeber et al. "PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes". In: *High Performance Computing, Networking Storage and Analysis, SC Companion:* Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 1116–1123. ISBN: 978-1-4673-3049-7. DOI: 10.1109/SC.Companion.2012.134. URL: http://dx.doi.org/10.1109/SC.Companion.2012.134.