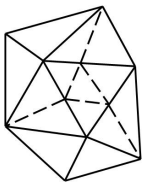

Imperial College London

Department of Computing

Anisotropic mesh coarsening and refinement on GPU architecture

Author:
Matthew Potter

Supervisors:
Gerard Gorman and
Paul H J Kelly

Tuesday 21st June, 2011
www.doc.ic.ac.uk/~mjp07

Abstract

Finite element and finite volume methods on unstructured meshes offer a powerful approach to solving partial differential equations in complex domains. It has diverse application in areas such as industrial and geophysical fluid dynamics, structural mechanics, and radiative transfer. A key strength of the approach is the unstructured meshes flexibility in conforming to complex geometry and to smoothly vary resolution throughout the domain. Adaptive mesh methods further enhance this capability by allowing the mesh to be locally modified in response to local estimates of simulation error. The ability to locally control simulation error plays an important role in both optimally exploiting available computational resources to achieve the most accurate solution feasible, or simulating a process to within design/safety guidelines for example.

This report focus on the anisotropic adaptive mesh operations of coarsening and refinement on meshes of 2D simplexes. The novelty of this work centres on recasting the coarsening and refinement algorithms, which were developed for serial execution on CPU's, into a form better suited to the massively parallel nature of GPU's. An impressive speedup has been achieved when compared against the best known multi-threaded CPU implementation on top of the range hardware.

Acknowledgements

Many thanks to the following people to whom I am extremely grateful for their help and support.

- Professor Paul H J, who initially recommended a project in high performance computing and introduced me to mesh adaptivity. His supervision, feedback and guidance has been invaluable during the course of this project.
- Dr Gerard Gorman, an expert in the field of mesh adaptivity. He provided a wealth of practical knowledge in designing and implementing mesh adaptivity algorithms. He pointed me in the right direction a number of times and pointed out many things which I had failed to consider.
- Georgios Rokos, a PhD student working on mesh adaptivity. Georgios pioneered mesh adaptivity on the GPU for his MSc thesis. This work was the starting point of the project. Georgios's help and knowledge gained from first hand experience was undoubtedly key to the success of the project.

Contents

Glossary

acceptable mesh A mesh which is acceptable according to some error metric. 29

anisotropic coarsening A coarsening technique which splits an element up into smaller elements that do not necessarily have the same shape. 57

isotropic refinement A refinement technique which splits an element up into smaller elements with exactly the same shape as the original. 29, 57

mesh A mesh is an connected undirected graph. 1

undirected graph A collection of nodes joined by edges, these edges do not specify a direction. vi

valid/conformant mesh A valid or conformant mesh only contains simplexes where all vertices belong to one or more elements. 31

Acronyms

APU accelerated processing unit. 59

CFD computational fluid dynamics. 1

DMA direct memory access. 44

FEM finite element method. 1

HPC high performance computing. 28

MPI message passing interface. 27

PBS portable batch system. 41

PDE partial differential equation. 1, 13

Chapter 1

Introduction

In order to solve problems in the field of computational fluid dynamics (CFD) one popular technique is the finite element method (FEM) . FEM solves partial differential equations (PDEs) across an element. The space in which the problem lies is split into many elements, the combination of these elements is called a mesh, FEM is then applied to each of these elements. The size and shape of these elements is vital in producing an accurate result and computationally efficient procedure. With a fine mesh of many small elements the result will be accurate, but at high computational cost. Similarly, a coarse mesh of few large elements will be computationally cheap, but will yield a poor result. The solution is to tailor the mesh to the particular problem, with many small elements in areas of high volatility and few large elements in areas with little change. This is where mesh adaptivity comes in.

Mesh adaptivity is a process of locally altering a mesh to maintain solution error estimates to within user specified bounds. Mesh adaptivity not only attempts to achieve a good result by altering the size of elements within the mesh, but also their shape and orientation. This thesis presents an investigation into techniques for coarsening and refining, and their suitability for execution on highly parallel architecture, mainly Nvidia graphics cards.

The very nature of coarsening and refinement is difficult for GPU computation. Coarsening and refinement involve topographical changes to the mesh in an irregular manner. So far much of work done on GPU programming has been of easily scalable, structured data computation done in a regular manner. The suitability of GPUs for more complex tasks like this has not really been investigated. This thesis explores this challenge and also provides some useful insight for future work. Mesh adaptivity is expensive; this work has demonstrated a substantial performance improvement through using a manycore GPU accelerator processor.

1.1 Motivation and objectives

In CFD, FEM on an unstructured adaptive mesh is one of the more complicated and exotic methods. It is not an easy problem to tackle and furthermore it is not something which is simple to implement on a GPU, as it goes beyond much of what has done before in GPU computation. For the many advantages GPU computation can offer there are some limitations, most of these limitations will be explored and solutions to overcome them in this particular problem.

1.2 Contributions

The completion of this project has led to the following notable contributions:

- High level of performance when compared against optimised multi-threaded CPU code of best known performance on a 12 core Westmere-EP. A speed up 40 times was achieved for a single pass of heavy coarsening on medium and large sized meshes. A speedup of 55 times or more was achieved for a single pass of heavy refinement on medium and large sized meshes. Average adaption time of just 0.18 micro-seconds per facet on a large mesh. Moreover 50% of the execution time is accounted for by memory transfer to and from the GPU, therefore the application is running within 50% of the theoretical maximum performance of an algorithm that takes 0 time.
- Successful design and implementation of solutions to unstructured problems, somethings which has previously been considered unsuitable for GPU computation due it the irregular memory access and high degree of branching.
- Systematic evaluation of performance covering many aspects of CUDA programming including thread divergence, coalesced memory access, pinned memory, asynchronous memory transfer, occupancy and L1 cache.
- Adaptation of parallel coarsening and refinement algorithms to utilise highly parallel architecture which avoids the need to maintain and colour large adjacency graphs. In the case of refinement only facet adjacency graph needed to be coloured. In the case of coarsening colouring was removed all together in favor of on the fly calculation of dependent operations.
- Decomposition of large serial tasks into smaller tasks which can be executed independently in parallel instead of the previously explored technique of larger tasks that achieve parallelisation through the use of thread communication.

1.3 Previous work

Mesh adaptation is a highly complex field, with many different techniques and many different approaches and algorithms for these techniques. All of this will be surveyed in chapter 2.

The Applied Modelling and Computation Group at Imperial College London [1] has developed a CFD application called "'Fluidity"'. Fluidity is an open source, general purpose, multi-phase computational fluid dynamics code capable of numerically solving the Navier-Stokes equation and accompanying field equations on arbitrary unstructured finite element meshes in one, two and three dimensions [2]. Fluidity is used in a number of different scientific areas including geophysical fluid dynamics, computational fluid dynamics, ocean modelling and mantle convection.

The complexity of modern CFD problems has lead to the need for thousands of processors and many days of computation. Due to this an effort has been made to implement mesh adaptivity on GPUs. The first stage of this effort has been completed by Georgios Rokos is his MSc Thesis where 2D mesh smoothening has been implemented in CUDA [Rok10a]. This project continues directly on from this work.

1.4 Statement of originality

This report represents my own work and to the best of my knowledge it contains no materials previously published or written by another person for the award of any degree or diploma at any educational institution, except where due acknowledegment is made in the report. Any contribution made to this research by others is explicitly acknowledged in the report. I also declare that the intellectual content of this report is the product of my own work, except to the extent that assistance from others in the projects design and conception or in style, presentation and linguistic expression is acknowledged.

1.5 Report outline

The remainder of the report is organised as follows: Chapter 2 and 3 gives a comprehensive description of the main principles and algorithms that govern the topic of refinement and coarsening of unstructured meshes. More precisely, chapter 2 gives an overview of graph colouring, a topic which is essential for parallel execution of mesh adaptivity; the quality of a mesh, particularly the local quality, and how this is evaluated and compares the tools available for the development of highly parallel programming. Chapter 3 is a detailed look at refinement and coarsening methods, including algorithms. Chapter 4 looks at any related work that have been carried out prior or during the time frame

[1] http://amcg.ese.ic.ac.uk
[2] http://amcg.ese.ic.ac.uk/Fluidity

of this project. Chapters 5 and 6 describes the design choices and the actual implementation of the target application, the data structures used, and any points of interest that were key to realising the project objectives. Chapter 7 presents performance optimisation made as well as a detailed analysis of the implementation, a look at the scalability of the application and a comparison against the best known mulit-threaded CPU equivalent. Finally, Chapter 8 summarises the main concepts, achievements, the experience gained throughout this project and lists the topics that remain open for further study and future work.

Chapter 2

Background

This chapter will cover the background work to the project. Issues in graph colouring have been explored as well as a few important graph structures. Tool and programming languages used for parallel computation on GPUs has also been studies and evaluated for use in the project.

2.1 Graph colouring

Graph colouring is performed to segregate the mesh into independent sets that can be updated concurrently without creating data conflicts. To achieve this each vertex in the graph is assigned a colour such that no adjacent vertices have the same colour.

Different types of mesh adaptivity require different graphs to reflect data dependence. Although the graph being coloured may represent different things, the algorithm used to colour it can be the same. Described below are two methods for graph colouring.

2.1.1 First fit colouring

First fit colouring, also known as greedy graph colouring considers the vertex of a graph and assigns the first available valid colour to that vertex, creating a new colour when required (Algorithm 1). First fit colouring often produces results which are far from optimal (where an optimal colouring is a colouring that uses the least number of colours) [WP67].

Algorithm 1 First fit colouring algorithm - Given a graph G(V, E) with vertex set V = (v_1, . . . , v_n) and adjacency lists A_j find colours c(v_j)

for $j = 1 \rightarrow n$ **do**
  $c(v_j) \leftarrow 0$
end for
$c(v_1) \leftarrow 1$
for $j = 2 \rightarrow n$ **do**
  $c(v_j) \leftarrow min(k \in \mathbb{N} | c(w) \neq k \forall w \in A_j)$
end for

With first fit colouring (Figure 2.1) the graph was coloured by traversing the graph in node order (a, b, c, d, e), this resulted in using 4 colours. One way of improving the outcome of first fit colouring is to carefully pick the order in which the graph is traversed. Using the same algorithm as before, but traversing the graph (b, c, e, a, d) we only use 3 colours (Figure 2.2). This now poses a new problem, how do we determine a suitable order such that we get good results. A randomly chosen order has a high probability of producing poor results for certain graphs. [BLS99]

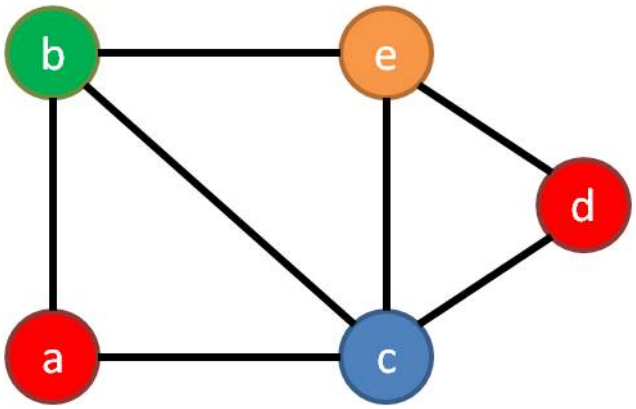

Figure 2.1: First fit graph colouring, coloured in order a, b, c, d, e

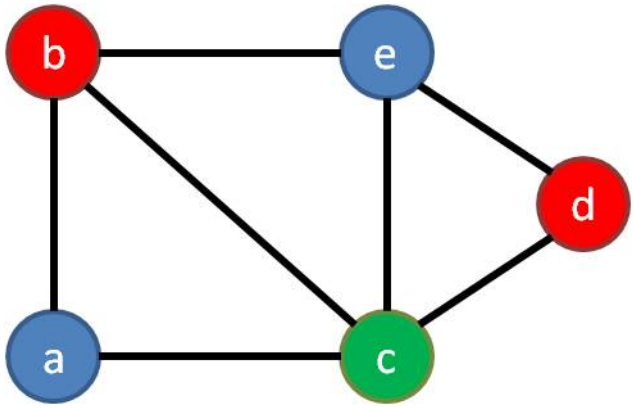

Figure 2.2: First fit graph colouring, coloured in order b, c, e, a, d

The vertices of any graph may always be ordered in such a way that first fit colouring produces an optimal colouring. Given a graph of optimal colouring, one may order the graph by the colours. Then when one uses a greedy algorithm with this order, the resulting colouring is automatically optimal. This is however a trivial solution, as you need to know the optimal colouring first, making this approach redundant. The graph colouring problem is NP-Complete, therefore it is difficult to find an ordering that leads to a minimum number of colours. For this reason, heuristics have been used which attempt to reduce the number of colours but not necessarily guaranteeing an optimal solution. According to Brooks' theorem a graph such that the maximum number of adjacent vertices to any given vertex is Δ, the graph can be coloured with Δ colours except for two cases, complete graphs and cycle graphs of odd length, for these $\Delta + 1$ colours are needed. Brooks' theorem is used to determine the chromatic number (the smallest number of colours needed to colour a graph) of a graph. Using this, we can evaluate the colouring [Bro41].

A popular ordering is to choose a vertex V of minimum degree (least number of adjacent vertices), order the remaining vertices, and then place V last in the ordering. With this ordering it will use at most $\Delta + 1$ colours, and is therefore at worst one worse than Brooks' colouring [Chv84].

2.1.2 Multi-level colouring

Multi-level colouring is an approach used to colour graphs in parallel. The first step is to partition the graph. The graph connecting these partitions is then coloured to form a top level colouring. Each partition or sub graph is then coloured using a set of colours unique to the colour of that partition, these sub graphs can be coloured concurrently. Multi-level colouring requires far more colours than first fit colouring, but can be performed in a parallel manor. The main issue with multi-level colouring is partitioning the graph, a non-trivial problem [Wal01].

2.1.3 Jones-Plassmann colouring

Jones-Plassmann colouring algorithm is another parallel colouring technique (Algorithm 2). Initially every node in the graph is assigned a random number. In parallel every node who's random number is higher than all other uncoloured adjacent nodes is coloured with the first available colour. This is repeated until all nodes have been coloured [JP95].

Algorithm 2 Jones-Plassmann colouring, V being the set of all vertices in the graph

```
U ← V
while (|U| > 0) do
  for all vertices v ∈ U do in parallel do
    I ← (v such that w(v) > w(u) ∀ neighbors u ∈ U)
    for all vertices v' ∈ I do in parallel do
      S ← (colours of all neighbors of v)
      c(v') gets minimum colour not in S
    end for
  end for
  U ← U - I
end while
```

2.1.4 Graph structures

There are several important graph structures required for mesh adaptivity [Rok10b]. Below is a description of the graph structured referred to in this report.

Vertex graph The vertex graph is the graph joining vertices. In essence the mesh is defined as the vertex graph (Figure 2.3).

Element graph The element graph joins every adjacent element. In the case of 2D every internal element in the graph has three neighbours, every boundary element has two and the corner element have one (Figure 2.4).

Vertex-Element graph The vertex-element graph joins an element to the three vertices which is consist of. Every element will be connected to exactly three vertices, each vertex is connected to any number of elements greater than two (Figure 2.5).

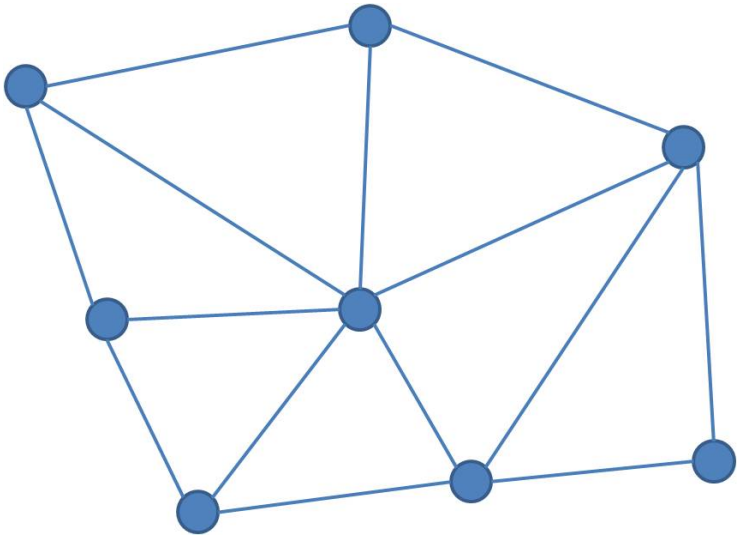

Figure 2.3: Vertex graph on mesh

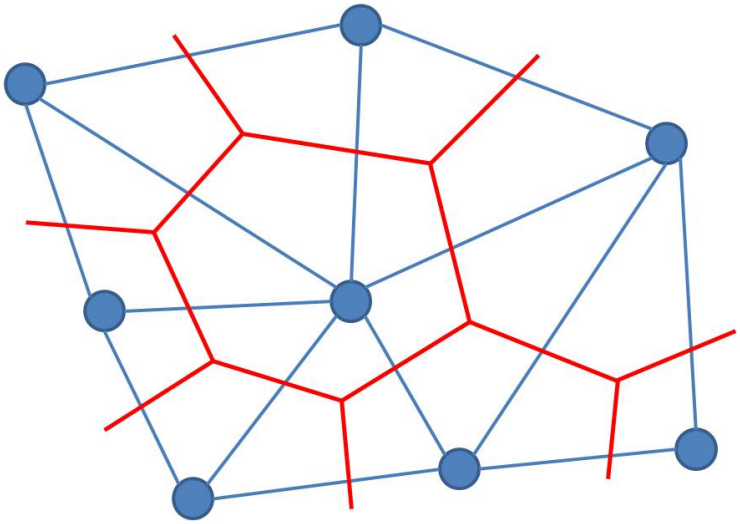

Figure 2.4: Element graph (shown in red) on mesh (shown in blue)

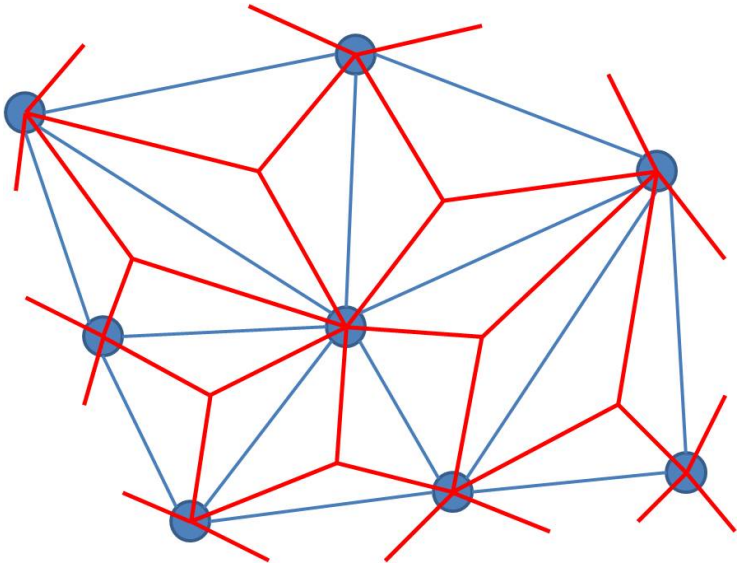

Figure 2.5: Vertex-element graph (shown in red) on mesh (shown in blue)

2.2 Evaluating the mesh

Mesh adaptivity is a heuristic approach at altering a mesh to be not only conformant but also quality determined by some metric. Various types of metrics can be used depending on the application and the type of adaptivity method used. The quality of a mesh is evaluated against the metric tensor in metric space and not in mesh space. Below are a few common quality metrics.

Size of element angles The size of the smallest angle in an element (the largest the smallest angle the better). This metric is often used when evaluating whether to flip an edge or not.

Length of longest edge The length of the longest edge (the smaller the better). Often used for regular refinement.

Length of shortest edge The length of the shortest edge. Used to determine whether or not to remove an element via coarsening.

Element side The area (2D) or volume (3D) of an element.

Lipnikov functional Proposed by Vasilevskii and Lipnikov [VL99] this takes both element size and shape into account. This is used for smoothening.

2.3 GPU architecture

This section is brief as there is plenty of good sources documenting GPU architecture, most of which can be found in CUDA Zone [1]. The main challenge of GPU programming is to fully utilise the GPUs many small processors. This can be achieved by launching hundreds if not thousands of threads. GPUs are also subject to more limitations than CPU and these limitations need to be well understood before an successful application can implemented for a GPU. Every thread in a warp (32 thread block) must either execute the same instruction or no instruction at all, this means that code with a large amount of control flow can be difficult to implement on the GPU. Another factor worth noting is the cost of transfer data to and from the GPU. The GPU cannot access main memory, therefore any data to be used by the GPU must first be transfer there by CPU. For more information please consult the CUDA programming guide [2].

2.4 Tools for Parallel Computation

It is important to select the right language a tools for any software development. Intelligent selection will help satisfy performance objects as well as reduce effort and increase the chance of a successful outcome. Presented here is a selection of the main options considered, evaluating each one.

2.4.1 OP2

OP2 is an open-source framework for the execution of unstructured grid applications on clusters of GPUs or multi-core CPUs. Although OP2 is designed to look like a conventional library, the implementation uses source-source translation to generate the appropriate back-end code for the different target platforms. OP2 continues on from the OPlus library which was developed more than 10 years ago. The main motivation behind OP2 is to handle multi core architectures [3].

OP2 uses sets to describe unstructured grids; these sets could represent different information, either nodes, vertices, elements, edges etc. Associated with these sets are both data and mappings to other sets. All of the numerically-intensive operations can then be described as a loop over all members of a set, in this way you define something similar to a CUDA kernel, a function which is executed for every iteration of the loop. With the limitation that the ordering of which this function is applied does not affect the result, OP2 can parallelise the execution.

When an application is written in OP2 it can then be built into three different platforms: single threaded CPU, parallelised using CUDA for NVIDIA GPUs or multi-threaded using OpenMP for multi core x86 systems. This ability to

[1] http://www.nvidia.co.uk/object/cuda_home_new_uk.html

[2] http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

[3] http://people.maths.ox.ac.uk/gilesm/op2/

run the same code on different platforms could be extremely useful for this application. It gives the developer the ability to compare the performance of different aspects of the application on different architecture, without the need to write new code.

The major drawback to OP2 is that you cannot have dynamic sets. If the dimensions of a set are going to change, you have to re-declare that set (much like an array in C). This makes any topological changes to the mesh very inefficient in OP2, which is the reason why it is not a good candidate for this kind of mesh adaptivity, despite its many strengths.

2.4.2 Galois

The Galois project aims to make it easy to write data-parallel irregular programs, and then efficiently execute them on multi core processors. Galois has a programming model consisting of [4]:

- Two simple syntactic concurrency constructs
- An object model which allows for increased concurrency during parallel execution
- A run-time system which supports the concurrency constructs and object model

Using the Galois system, a range of data-parallel irregular applications have been parallelised, achieving significant speedups on algorithms which appear to have too many sequential dependences to be parallelisable.

2.4.3 STAPL

STAPL (Standard Template Adaptive Parallel Library) is a C++ framework for developing parallel programs. Its core is a library of ISO Standard C++ components with interfaces similar to the ISO C++ standard library. STAPL includes a run-time system, design rules for extending the provided library code, and optimization tools [RAO98].

STAPL aims to improve programmability and portability of performance in parallel application. Programmability refers to abstracting away specifics of parallel algorithms from the developer, making development of parallel application easier and quicker, this is the goal STAPL shares with all the other frameworks mentioned so far. Portability of performance is the idea that you can execute the same code on different architectures and not suffer from performance degradation. Parallel algorithms are generally very sensitive to platform architecture.

STAPL is divided into three levels of abstraction; the level required by the developer depends on his needs, experience and time available.

[4] http://iss.ices.utexas.edu/galois.php

Level 1 Application Developer

Level 2 Library Developer

Level 3 Run-Time System (RTS) Developer

The highest level of abstraction, Application Developer, STAPL gives the developer interfaces to building blocks for parallel programs which they can piece together to build an application. At this level no great understanding of parallel programs is required. For a more experienced developer the next level (Library Developer) offers greater flexibility and control. The developer can define his own library functions either to extend the existing functionality or add domain specific operations. The majority of STAPL development is done in these two layers, but for greatest amount of control over the execution of parallel programs a low level layer was added, the RTS Developer layer. In this layer you have access to the implementation of the communication and synchronization library, the interaction between OS, STAPL thread scheduling, memory management and machine specific features such as topology and memory subsystem organization.

2.4.4 Liszt

Liszt is a domain specific language developed at Stanford University, designed specifically for mesh based PDEs problems. Liszt can be complied into a variety of different implementations including CUDA, OpenMP and MPI and promises to add additional implementations in the future. Liszt code is written at a high level, the abstraction allows the Liszt compiler to aggressively optimise the code and automatically decide how to partition the mesh. The compiler is also capable of changing the layout of memory to suite a particular architecture. The project claims to offer many useful features, but is not yet complete so therefore not available for this project [5].

2.4.5 X10

Although X10 is not specifically designed for use with unstructured meshes, it is worth while looking into. The above are all frameworks for parallel programs whereas X10 is a fully type safe, parallel orientated programming language. X10 is being developed by IBM and although it is a fairly new project, it is quickly advancing. X10 not only aims to be able to write parallel applications easily, but also distributed ones [6].

2.4.6 CUDA

CUDA (Compute Unified Device Architecture) is developed by Nvidia to be an accessible GPU computing engine for Nvidia graphics cards. Developers use

[5] http://ppl.stanford.edu/wiki/index.php/Liszt
[6] http://x10-lang.org/

"'C for CUDA"' which adds CUDA extension to the C (and C++) language. These extensions allow developers to transfer data to and from device memory (the device being the Nvidia graphics card) as well as execute code, known as kernels, on the device. The vast majority of current GPU acceleration work has been done using CUDA due to it's flexibility and maturity when compared with the alternatives [7].

2.4.7 OpenCL

Recently companies including Apple, Intel, AMD/ATI and Nvidia, have jointly developed a standardised programming model, OpenCL. [8]. Much like CUDA, OpenCL defines a set of C/C++ language extensions which allow development for highly parallel device code. OpenCL is a royalty free, open standard for cross platform GPU programming. The reason why OpenCL has not yet overtaken CUDA as the prominent GPU programming technology is because OpenCL is still much more tedious to use, also the speed achieved in an application written using OpenCL is much lower than the equivalent application written using CUDA. GPU programming is primarily motivated by speed, therefore we expect the majority of people to choose the most per formant technology [KmWH10].

2.5 Summary

The initial necessary background has been explored as well as appropriate tools this project. The next chapter will look into the algorithms that have been proposed for mesh coarsening and refinement.

[7] http://www.nvidia.com/object/cuda_home_new.html
[8] http://www.khronos.org/opencl/

Chapter 3

Mesh adaptivity algorithms

This chapter provides a literary review of a wide range of papers on mesh adaptivity for coarsening and refinement. In particular close attention has been put towards parallel algorithms for mesh adaptivity.

Mesh adaptivity can be separated into two main categories, topological adaptivity and non-topological adaptivity. Non-topological adaptivity is achieved by smoothening, a process of relocating vertices within a mesh. Topological adaptivity is achieved by refining, coarsening and edge flipping. This work focuses on coarsening and refinement. Various algorithms for refinement and coarsening are described in this chapter. A combination of these topological and non-topological techniques are used to alter the mesh and converge to an acceptable result. One such way of combining various adaptivity methods to achieve a suitable mesh is given in Algorithm 3 which was first proposed by Freitag et al [FJP98].

3.1 Refinement

Mesh refinement is the process of increasing resolution locally to a mesh. Refinement methods can be divided into two types, non-hierarchical and hierarchical. Hierarchical refinement can be represented in a tree structure or hierarchy. The top level of the mesh contains the original elements, additional levels are then added to any element which is not yet good enough (Figure 3.1). Non-hierarchical methods cannot be represented in such a way.

Algorithm 3 Framework for adaptive solutions of PDEs - Given a vertex set V = (v_1, . . . , v_n) and element set T = (v_1, . . . , v_m)

$k \leftarrow 0$
Construct and initial mesh, M_0
Improve M_0, using flipping and smoothening, to form $M_0^{'}$
Solve PDE on $M_0^{'}$
Estimate the error on each element
while the maximum error on an element is larger than the give tolerance **do**
  Based on error estimated, determine a set of elements, S_k, to refine
  Divide the elements in S_k, and any other element necessary to form M_{k+1}
  Improve M_{k+1}, using flipping and smoothening, to form $M_{k+1}^{'}$
  Solve the PDA on $M_{k+1}^{'}$
  Estimate the error on each triangle
  $k \leftarrow k + 1$
end while

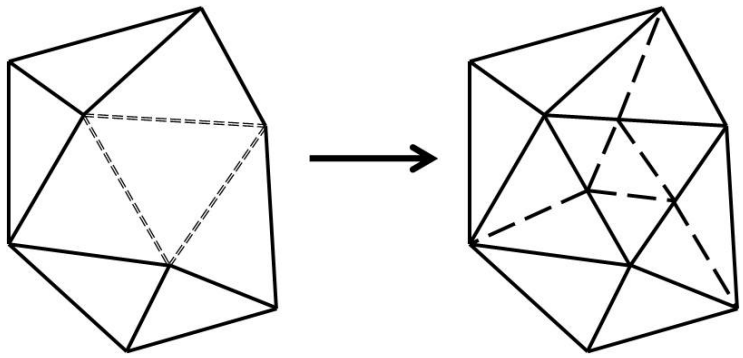

Figure 3.1: Mesh before hierarchical refinement (left) Mesh after hierarchical refinement on marked element (right)

3.1.1 Non-hierarchical methods

Bisection

Bisection is the process of dividing either an element in two or bisecting an edge. These two approaches yield the same result, but the process is different. Element bisection involves iterating over each element and evaluating firstly whether the element is good enough; if it is not then you evaluate where to bisect the element. There are three possible cases in 2 dimensions (Figure 3.2). When an element has been bisected a non-conforming mesh is created, this is because the neighbouring element contains four vertices. To rectify this, the bisection needs to be propagated, how the bisection is propagated depends on

how many neighbouring elements have been bisected (Figure 3.3).

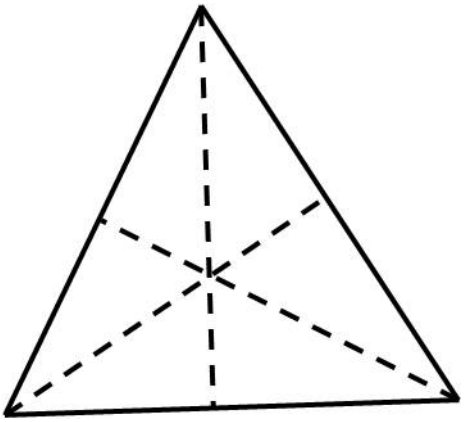

Figure 3.2: Three possible element bisections

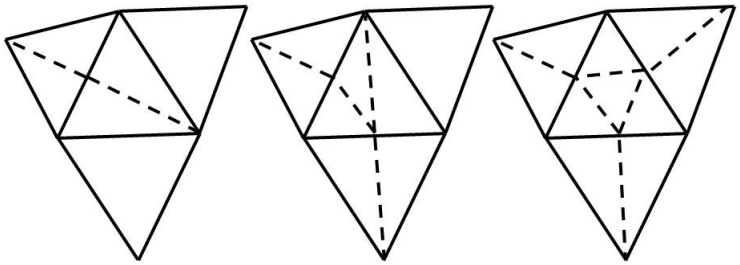

Figure 3.3: Bisection propagation

The process of propagating element bisections to achieve a conforming mesh is very complicated as a lot of mesh connectivity information needs to be kept and maintained (you need an edge element adjacency list, and element element adjacency list as well as vertex information). All of this can be computationally expensive. Edge bisection avoids the need to propagate bisections.

Performing edge bisection involves iterating over edges and deciding whether to bisect the neighbouring elements. If an edge is bisected, the un-bisected edges of the neighbouring elements need to be updated, as two of the four edges will now be adjacent to a new element (Figure 3.4).

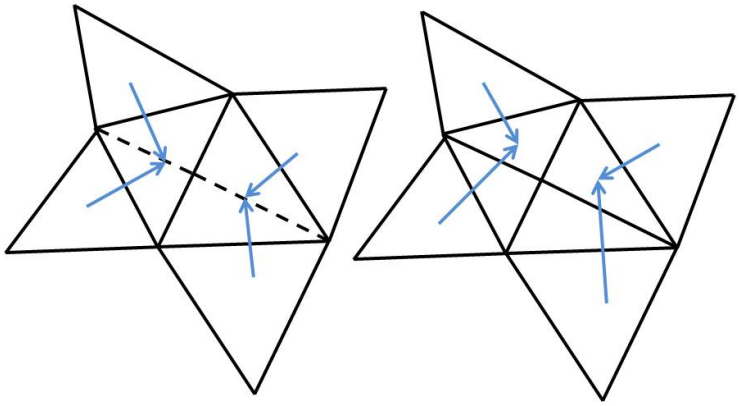

Figure 3.4: Edge bisection, arrows show adjacent elements

Rivara described an algorithm for bisection, see Algorithm 4. Initially elements are marked for refinement based on a quality metric, non-conforming elements created by the algorithm are subsequently marked for refinement (this is how the algorithm propagates the bisection). The algorithm continues until a conforming mesh has been constructed [Riv84].

Algorithm 4 Rivara's longest edge bisection algorithm

Let T_0 be the set of marked elements
$i \leftarrow 0$
while $T_i \neq \emptyset$ **do**
  Bisect elements in T_i across their longest edge
  Let T_{i+1} be the set of nonconforming elements
  $i \leftarrow i + 1$
end while

Rivara proves that this algorithm will terminate, however no useful bound exists for the number of times the while loop is executed [JP97a]. To resolve this Rivara proposed variants of this algorithm including one which propagates the longest edge bisection with a *simple bisection* [Riv84]. Simple bisection is a bisection across any edge, not necessarily the longest. In this variant the marked elements are first bisected along their longest edge, if any non-conforming elements are created, a simple bisection occurs (Algorithm 5). This algorithm is used for the first refinement set, further steps must assign elements to V_i and T_i based on whether the element originated from a longest-edge bisection or not.

Algorithm 5 Rivara's modified bisection algorithm

Let T_0 be the set of marked elements T will denote elements not yet refined
$V_0 \leftarrow \emptyset$ V will denote children of refined elements
$i \leftarrow 0$
while $(T_i \cup V_i) \neq \emptyset$ **do**
  Bisect elements in T_i across their longest edge
  Bisect elements in V_i across a non-conforming edge
  Let V_{i+1} be the set of non-conforming elements embedded in $\cup_{j=0}^{i} T_j$
  Let T_{i+1} be the set of all other elements
  $i \leftarrow i + 1$
end while

Parallel bisection

In order to execute bisection in parallel the usual method would be to colour the mesh such that bisection of an element in a colour does not affect other elements in that colour. This is the approach taken when parallelising other adaptivity algorithms. However, in practice, the overhead associated with maintaining the colouring outweighs the advantages of parallel computation. In response to this Jones and Plassmann developed PRAM [1] adaptive refinement algorithm that avoids the synchronisation problems that make maintaining colouring ineffective (Algorithm 6). Synchronisation problems are avoided by simultaneously refining element from independent sets. The independent sets used for refinement are also used to update the colouring, this is required because the dual graph is modified after the bisection of an element. Jones and Plassmann proved the below algorithm avoids all possible synchronisation problems and has a fast run time [JP97b].

[1] Parallel Random Access Machine - a shared memory abstract machine used by parallel algorithm designers to model algorithmic performance (such as time complexity). The PRAM model neglects such issues as synchronization and communication, but provides any (problem size-dependent) number of processors. Algorithm cost, for instance, is estimated as O(time x processor number) [KKT01].

Algorithm 6 Jones and Plassmann's parrallel refinement algorithm

$i \leftarrow 0$
Let T_0 be the set of marked elements
Each element in T_0 is assigned a unique random number, $\rho(t_j)$
The subgraph DT_0 is coloured
$\mathrm{V}_0 \leftarrow \emptyset$
while $(T_i \cup V_i) \neq \emptyset$ **do**
  $W_i \leftarrow T_i$
  while $(T_i \cup V_i) \neq \emptyset$ **do**
    Choose an independent set in D, I, from $(T_i \cup V_i)$
    Simultaneously bisect elements in I embedded in T_i across their longest edge
    Simultaneously bisect elements in I embedded in V_i across a non-conforming edge
    Each new element, t_j, is assigned the smallest consistent colour, $\sigma(t_j)$, and a new processor
    Each processor owning a bisected element updates this information on processors owning adjacent elements
    $V_i \leftarrow V_i\ (I \cap V_i)$
    $T_i \leftarrow T_i\ (I \cap T_i)$
  end while
  Let V_{i+1} be the set of non-conforming elements embedded in $\cup_{j=0}^{i} W_j$
  Let T_{i+1} be the set of all other elements
  $i \leftarrow i + 1$
end while

Jones and Plassmann went on to present a more practical version of the above algorithm that rather than assigning a single element or vertex to each processor, a set of vertices and elements is assigned to each processor. This algorithm will not be shown in this report, but can be found in Jones and Passmann's paper "'Parallel Algorithms for Adaptive Mesh Refinement"' [JP97b].

3.1.2 Hierarchical methods

The most common type of hierarchical refinement is regular refinement. With regular refinement any element which requires refinement is split into four new elements (Figure 3.5). This now adds four new nodes below the element in the hierarchy. The process is repeated until a satisfactory level of refinement in each element is achieved (Figure 3.6). Much as in bisection, the changes are propagated to neighbouring elements to remove any non-conforming element. To prevent the need to propagate multiple bisection on the same edge, the condition of requiring all neighbouring elements to be at most one level of refinement away is added.

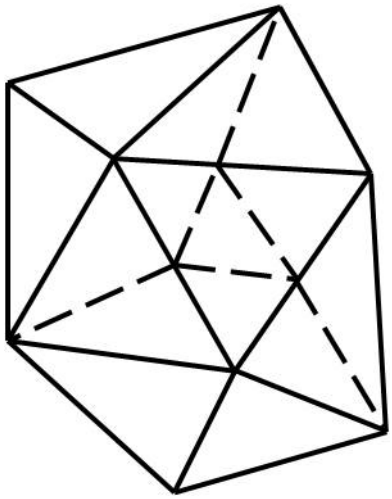

Figure 3.5: Regular refinement

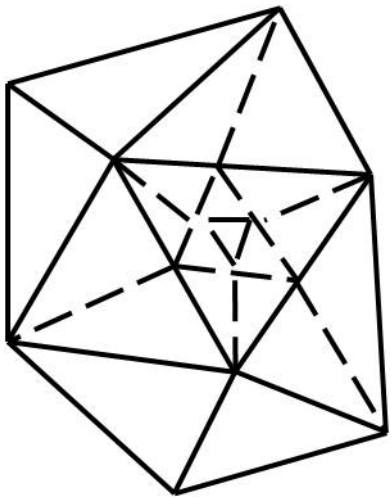

Figure 3.6: 2 level regular refinement

Bank et al worked on developing an algorithm for regular refinement (Algorithm 7). Initially elements are marked for refinement based on some metric. These elements are refined using regular refinement. This is then propagated by regularly refining all elements with at least two non-conforming edges and bisecting all elements with one non-conforming edge. Before the next refinement step, any bisected elements are merged [BSW93]. The resulting elements are exactly the same shape (yet a different size) to their parent (except for the elements bisected to create a conforming mesh), this means that this form of

refinement is isotropic.

Algorithm 7 Bank's regular refinement algorithm

All bisected elements are merged
Let T_0 be the set of marked elements
$i \leftarrow 0$
while $(T_i \cup V_i) \neq \emptyset$ **do**
  Regularly refine elements in T_i
  Let T_{i+1} be the set of elements with at least two non-conforming edges
  $i \leftarrow i + 1$
end while
Bisect remaining non-conforming elements across a non-conforming edge

3.2 Coarsening

Whereas mesh refinement locally increases the resolution of the mesh, coarsening reduces the local resolution of the mesh. There are several methods for mesh coarsening, in general the method of mesh coarsening is dependent on the method of mesh refinement as complementing refinement and coarsening methods will help reduce complexity and computational cost.

3.2.1 Reversing hierarchical refinement

This method only works with hierarchical refinement for obvious reasons. On the face of it reversing hierarchical refinement seems simple, just remove all nodes below a particular point on the tree representing the mesh, but because bisections are propagated more needs to be done. One of two things can be done to combat this. In the case that the neighbouring element is at a higher level or refinement and is not eligible for coarsening, then the current element needs to be bisected in the same way as discussed in bisection propagation. If the neighbouring element is of the same level of refinement as the current element and is simply bisected, then this bisection can be removed (in the case that it is bisected more than once, then the number of bisections can be reduced by one (Figure 3.7). A limitation of this method is that you can only get as coarse as the original mesh and no further.

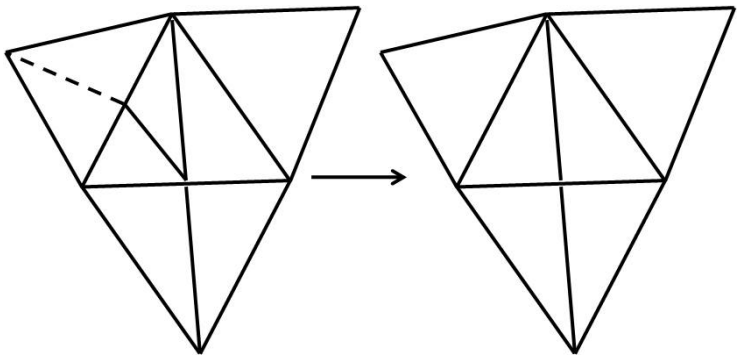

Figure 3.7: Removing bisection propagation

3.2.2 Edge collapse

Edge collapse involves reducing an edge to a single vertex and thereby removing two elements. Edge collapse can be used in conjunction with any refinement algorithm, but is best suited to edge bisection as it can be done in the same iteration over edges (although when used with regular refinement it can provide anisotropic adaptivity to an isotropic algorithm) and does not have to be a separate step, Figure 3.8 demonstrates edge collapse. Element element and edge element adjacency lists will need to be updated after an edge collapse. A disadvantage of edge collapse is that resulting vertex is of higher degree than either vertex of the now collapsed edge. This can pose problems in colouring, potentially increasing the number of independent sets and thereby reducing possible parallelisation in later optimisation steps.

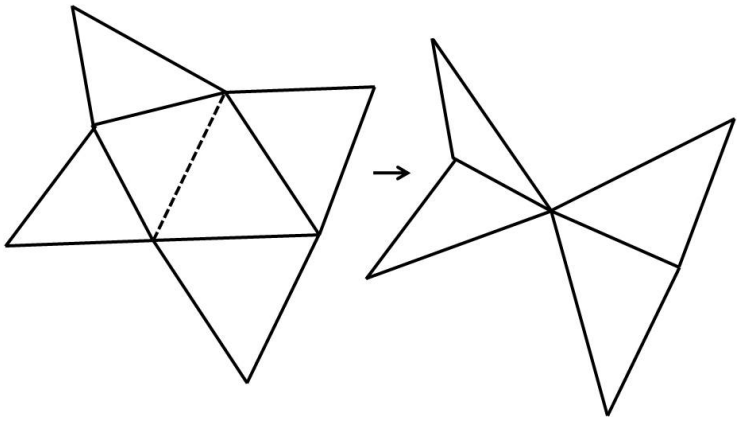

Figure 3.8: Edge collapse

Li, Shepard and Beall presented an algorithm for coarsening (Algorithm 8). This algorithm considers smoothening as a coarsening technique whereas for the

purpose of this report it is considered a mesh adaptivity technique of its own. The algorithms first initialises a dynamic vertex list with the end vertices of the given short edge list. A tag process if used to determine if a vertex is already in the list. It then iterates over this list and selects an appropriate coarsening technique to improve the mesh, using edge length as a quality metric (L_{low} being the lower bound for edges and L_{max} being the upper bound) [LSB05].

Algorithm 8 Li, Shepard and Beall's coarsening algorithm

```
for all edge in short edge list do
  for all vertex that bounds the current edge do
    if vertex is not yet tagger then
      append vertex to dynamic list
      tag vertex to be in dynamic list
    end if
  end for
end for
while vertices not tagged processed in dynamic list do
  get an unprocessed vertex V_i from the list
  get E_j, the shortest mesh edge in transformed space connected to V_i
  if the transformed length E_j is greater than L_low then
    remove V_i from the dynamic list
  else
    evaluate edge collapse operation of collapsing E_j with V_i removed
    if the edge collapse would create an edge longer than L_max then
      evaluate relocated vertex V_i
    else if the edge collapse would lead to flat/inverted elements then
      evaluate the swaps(s)/collapse compound operation
    end if
    if any local mesh modification is determined then
      tag neighbouring vertices of V_i in the dynamic list as unprocessed
      apply the local mesh modification
      remove V_i from the dynamic list if it is collapse
    else
      tag V_i as processed
    end if
  end if
end while
```

Parallel coarsening

Alauzet, Li, Seol and Shephard proposed an algorithm for parallel coarsening (Algorithm 9). The algorithm works in a similar manner to the one above but allows for parallel computation. To reduce the number of total mesh migrations, prior to each traversal, a buffer is initialised on each partition to store desired mesh modifications on partition boundaries [ALSS05].

Algorithm 9 Alauzet, Li, Seol and Shephard's parallel coarsening algorithm

```
while dynamic lists are not empty and there exists a vertex in the dynamic
list not tagged do
  for all vertices in dynamic list on each partition do
    determine a desired mesh modification operation to eliminate the current
    vertex
    if the element of the desired operation is fully on a partition then
      apply the operation, update the dynamic list and tag/untag vertices
    else
      put the desired mesh operation into the local buffer
    end if
  end for
  determine all requests to migrate mesh regions in terms of desired opera-
  tions in local buffers
  perform mesh migration and update each local dynamic list
end while
```

3.2.3 Element collapse

Element collapse is very similar to edge collapse except an element is reduced to a single vertex. This will remove four elements. Element collapse is best using with element bisection as it can be done in the same iteration. This method also suffers from the problem of increasing the degree of vertices (Figure 3.9).

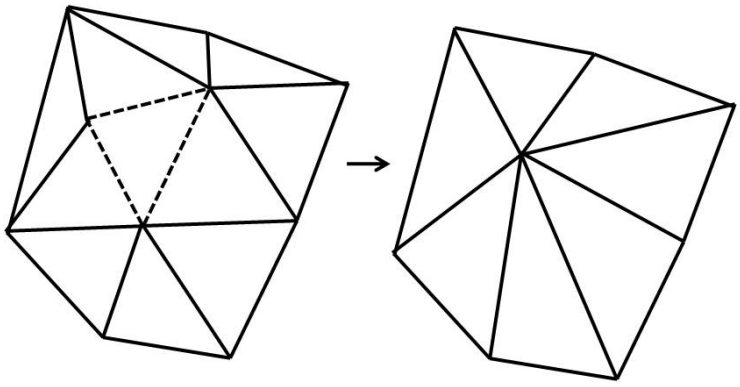

Figure 3.9: Element Collapse

3.3 Summary

Algorithms for coarsening and refinement have been explored, potential problems with the algorithms has also been noted. These algorithms help greatly in the design of this project, but do not provide any guidance in terms of imple-

mentation. The next chapter will look at related work in this field which will help in the actual implementation of this project.

Chapter 4

Related Work

Here we look at work that has been done, or is ongoing, that is shares common elements this project. Similarities and differences with this project are noted as well as any useful insights that has been gained from look at these projects.

4.1 Mesh adaptivity on the GPU

During the course of this project Timothy McGuiness at the University of Massachusetts Amherst has made some effort to implement some aspects of mesh adaptivity on CUDA. The work looks at smoothening, partitioning and graph analysis. Although the issues of coarsening and refining have not been visited by this project it does provide some helpful information. One important conclusion made from Timothy's work was that although GPU execution did provide some speedup over the CPU, it was overshadowed by the transfers times and Message passing interface (MPI). This project is not concerned with distributed computation, therefore MPI will not be a factor. The memory transfer times between the main memory and GPU will be [McG11].

4.2 Generic adaptive mesh refinement

Generic adaptive mesh refinement (GAMeR) technique was created in LaBRI-INRIA, University of Bordeaux. GAMeR is mesh refinement tool implemented on the GPU. It is a single pass algorithm which takes a coarse mesh from the CPU and uses pre-tessellated patterns to refine each element in the coarse mesh. The level of refinement is determined by a per-vertex scalar value which indicated the level of detail required near each vertex. GAMeR obviously is obviously concerned with coarsening and the approach to use pre-tessellated patterns simplifies matters considerable. The technique does work with any arbitrary initial mesh and is used for real time applications therefore executes in short periods of time. The success of GAMeR is promising and indicated that it will also be

possible to also gain good performance in this project [1].

4.3 HOMARD adaptive meshing

The HOMARD package performs adaptation for finite element or finite volume meshes with refinement and unrefinement. The technique they are using for refinement is the same as will be used in this project, regular refinement with edge splitting to for the purposes of conformity. Little is mentioned on the process of unrefinement, but this can be assumed to be the process of reversing refinement, meaning they store a hierarchy of refinement which can later be reversed. This package as also done work on implementing adaptivity in 3D as well as 2D, 3D mesh adaptivity is beyond the scope of this project, however the scaling of 2D to 3D can be estimated from results of HOMARD [2].

4.4 Summary

Work relating to this project has been reviewed. The next chapter explains the design of adaptivity application created for this project.

[1] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch05.html
[2] http://www.code-aster.org/outils/homard/menu_homard.en.htm

Chapter 5

Design

This chapter explores the design choices made and the justification behind those choices. The main concern with the design is the choice of algorithms and how these algorithms are put together to provide and complete coarsening and refinement solution. The type of error metrics and process of colouring is described. Issues with designing an application suitable for GPU computation are also considered. Finally we take a look at the main data structures for both CPU and GPU.

5.1 Design objectives

The aim of the project is to develop an application for performing mesh coarsening and refinement on the GPU. The main criteria considered were:

Scalability The primary application of this will be high performance computing (HPC), a design space which has scalability at its heart. The solution must have the ability to scale to larger problem sizes.

Comparable Implementations Both a multi-threaded CPU and GPU version must be implemented in order to compare results. Both versions must use the same adaptivity algorithms, although they may differ due to the differences in design spaces, in order to get the most out of each version, different approaches are taken.

Performance The motivation behind moving mesh adaptivity from the CPU to the GPU is to improve performance and execution time. The final solution must have a performance speed up when compared to the CPU implementation.

5.2 Algorithms chosen

The coarsening algorithm chosen is based on the parallel edge collapse described in Algorithm 9. After an extensive study of adaptive mesh algorithms, this one is the only one found that has been implemented in a parallel fashion, so it was an obvious choice.

The choice of refinement algorithms was a little more involved. Initially Jones and Plassmann's parallel refinement algorithm was chosen (Algorithm 6). This approach was rejected due to the additional need for creating maintaining edges, something which was avoided in coarsening (discussed in the next chapter). This algorithm is also designed to work with CPUs where each CPU core would do a large amount of work whereas a more suitable algorithm for the GPU would be where the work can be divided thousands of times so that each thread only does a small percentage of the overall work. Along with the coarse division of work, the algorithms relies on communication between threads, something with is best avoided in GPU computation.

After ruling out Jones and Plassmann's parallel refinement algorithm, Banks' regular refinement [BSW93] was chosen (Algorithm 7). Although there is no parallel version of this algorithm, it is easy enough to implement one (discussed in the next chapter). This algorithm does not need to create or maintain edges and can be implemented with one thread per element. The only issue with this algorithm is that it is an isotropic refinement and therefore will not do anything to correct badly shaped elements. This limitation is mitigated against with the use of anisotropic coarsening, the coarsening step will remove badly shaped elements while the refinement will improve the local resolution of the mesh. Iterating over these two steps will quickly converge to an acceptable mesh.

5.3 Error metric

As mentioned in the background, the choice of error metric is dependent on the method of adaptivity chosen. For edge collapse the only real concern is edge length. The process of collapsing an edge removes two elements that are ill formed. Using minimal edge length, ill formed is defined as having an element with an edge length less than the lower bound, this elements are considered slithers and can be removed. During the collapse it is important to be careful not to create elements which have an edge length greater than the upper bound, otherwise you may get into the situation where you are oscillating between coarsening and refining and never converging. Similarly for refinement, because the shape of the element cannot be altered as it is an isotropic refinement edge length is the only thing worth considering. If the maximal edge length is greater than the upper bound, then the element should be refined.

We now have an error metric that can be defined with an upper and lower bound for edge length, L_low and L_up. Choosing suitable values for these is problem dependent and should be set by the user, but it is useful to employ some constraints on these values in order to prevent oscillations occurring. This

very issue was invested by Shephard et at [ALSS05] equation 7, and as an outcome, the lower bound is always set as half the upper bound. $L_low = 0.5 * L_up$.

5.4 Adaptivity framework

The adaptivity framework controls what refinement techniques needs to be executed and in what order. It also performs the important job of terminating the adaptivity. The framework designed was based on work done by Shephard et al [ALSS05]. Algorithm 10 starts with an initial coarsening step, it then iterates over refinement and coarsening until either convergence or MAX_ITER loops, which would only occur if something has gone seriously wrong.

Algorithm 10 Basic Framework

```
Coarsen Mesh
for i = 1 → MAX_ITER do
  Refine Mesh
  Coarsen Mesh
  if Maximal mesh edge length < L_max then
    return Success
  end if
  i ← i + 1
end for
return Error
```

5.5 High level procedure overview

Choice of algorithms is only one part of designing a solution. Exactly how to execute these algorithms is a completely different problem. Which data structures to use, what information needs to be maintained and to alter the mesh in a consistent and concurrent manner all need to be considered.

The usual approach to parallel edge collapse and regular refinement is to divide the mesh into independent sets and then perform each edge collapse or regular refinement in parallel as one complete task. This is the approach taken for the CPU implementation. For the GPU implementation the edge collapse and regular refinement was broken down into many parts, each part was a separate CUDA kernel. This approach meant that only a very limited number of tasks needed to be performed as independent sets. It also achieved consistency of parallel computation by placing barriers at the end of each of these kernel invocations (discussed in more detail in the following chapter). Algorithm 11 is the high level coarsening algorithm. It first by marks every element which contains an edge with length smaller than L_low. Adjacent collapses are deferred if they are collapsing an edge larger than any adjacent collapse. This is done to

prevent data conflicts. Each marked collapse is then tested as to whether it is a valid collapse, a collapse is valid if it does not cause an element to invert and does not create an element with edge greater than $L_u p$. Next all valid collapses are actually performed followed by updating references to vertices now removed and updating of facet adjacency.

Algorithm 11 Coarsening Algorithm

```
repeat
    evaluate minimal edge length of all facets and mark facets for coarsening
    defer adjacent collapse that collapse larger edges
    remove invalid edge collapses
    collapse edge
    update removed vertices
    update facet adjacency
until mesh not changed
compact mesh
```

Algorithm 12 is the high level refinement algorithm, the lines marked with an asterisk are the tasks which need to be computed in an independent set. Tasks done on the CPU are also marked, all other tasks can be assumed to be executed on the GPU. First the facet adjacency information is calculated, then using this the mesh is coloured. Next every facet is evaluated as to whether it should be refined or not. The propagation of these marking are calculated to ensure a valid/conformant mesh, this propagation of markings continue until there is no change to the mesh. The indexes (described in the next chapter) are calculated for the new facets and vertices and then space is allocated for these new items. Finally the new facets and vertices are populated.

Algorithm 12 Refinement Algorithm

```
create facet adjacency on CPU
colour mesh on CPU
evaluate maximal edge length of all facets and mark facets for refinement
repeat
    propagate markings
until no change to markings
create new facet indexes
allocate space for new facets
mark vertices to be created*
create new vertex indexes
allocate space for new vertices
populate new vertices
populate new facets
```

5.6 Framework and data structures

The code base for this is an adaptation of the code based used in Georgios Rokos's smoothening MSc project. For a more detailed description of the initial framework please refer to Georgios's thesis [Rok10a]. Much as in the original code, the initial mesh is read in from a vtk file where the Vertices, Facets and Discrete Linear Metric are created (see below). The mesh adaptivity procedures are then called until convergence, i.e. no changes need to be made to to mesh in order for it to conform to the user set bounds.

5.6.1 CPU data structures

Vertex Describes a mesh node. 1 vector containing two co-ordinates (i and j).

Facet Describes a mesh element. 3 Vertex IDs, 3 Adjacent Facet IDs.

Metric Describes the metric tensor. A regular grid of metric values, the size of this grid depends on the inital mesh size.

Independent Sets A two dimensional array, the first dimension separating each independent set the second separating the IDs of Facets in the set.

5.6.2 GPU data structures

The GPU contains all the above data structures excluding independent sets.

Facet Colour Used as an alternative to independent sets. Lists the colour of the facets.

Marked Facets Represents whether a facet needs to be coarsened, regularly refined or bisected.

New Facet Index Stores the indexes of the newly created facets.

New Vertex Index Stores the indexes of the newly created vertices.

Vertices To Remove Marks the vertices to be removed during coarsening.

5.7 Summary

The design choices have been presented along with a high level overview of the solution. The reader should now have a sense of the overall application as far as algorithms and data structures are concerned. The next chapter will look in more detail at the implemented solution.

Chapter 6

Implementation

In this chapter the specifics of the implementation are described. The issues and challenges encountered are explored as well as how these were overcome. A novel approach has been taken to port this onto the GPU, the various techniques used for a successful port are described in detail.

6.1 Atomic operations

In the CPU implementation, the code will only be a handful of threads will be running concurrently, furthermore it is possible to communicate between all threads with the use of atomic operations, barriers and critical regions. As a result these devices were used, notably when a new facet or vertex needed to be created it would be added to a dynamic array, a data structure that has been optimised over the years. To protect against corruption from concurrent accesses, operations to this dynamic array were protected inside a critical block. Porting this over to the GPU posed several problems, the first of which is there is no dynamic array in CUDA so this had to be replaced with a standard array which would need to be resized [1]. Another limitation in CUDA is that you cannot communicate with any thread outside your block, the only way to do so is to use atomic operations on global memory, but because CUDA has hundreds if not thousands of threads running concurrently, you have to be very careful not to reduce your code down to a serial execution otherwise performance will suffer greatly.

It was clear that the first task would be to calculate the new memory requirements and then allocate this memory in one go. After some investigation it was also possible to calculate the IDs and therefore indexes of the new vertices and facets to be added. Every facet is first evaluated against the error metric and marked as to whether it needs to be refined or not. The propagation of this is then calculated by marking every element with at least two neighbours that are

[1] You cannot resize arrays in CUDA, instead an array of the new size is created and the old array is copied across.

being regularly refined for regular refinement, and marking elements with only one neighbour marked for regular refinement for bisection. This propagation set is iterated over until there is no change to the marking. From this marking we can calculate the memory requirements.

To calculate the memory requirements and new indexes an adaptation of a value reduction [2] algorithm was used. Value reduction is used to calculate things like the sum of an array. It works by first adding every pair together, then adding every pair of pair, continuing in this fashion until the whole array has been summed. With this never more than half the thread are active at any one time, it is a popular technique as it can be massively parallelised. This algorithm was adapted to create a running total from the array of marked elements. Every node marked for regular refinement would require three new facets and every node marked for bisection will require one. The marked elements array was marked with this in mind, if that element required bisection, it was marked with a one, if it required regular refinement it was marked with a three, for example [3,1,1,3,0,0], this means that elements 0 and 3 need to be regularly refined, elements 1 and 2 require bisection and elements 4 and 5 do not need to be altered. If a running total of this array was then created and then each item in the array was incremented by the current number of facets, you could then use this to assign unique IDs to the new elements. In the above example you would want to create an array like this, [8,9,10,13,13,13]. From this you could not only deduce that you would have 14 elements, but that the IDs for the three new elements needed to refine element 0 were 8, 7 and 6, likewise for the other elements.

Creating this running total was done by first, every odd item would be incremented by the previous element giving us [3,4,1,4,0,0] from the previous example. Then every item which is odd when divided by two (i.e. 2,3,6,7,10,11 etc) is incremented by the last element in the previous pair, giving us [3,4,5,8,0,0]. This is then repeated by every item which is odd when divided by four (i.e. 4,5,6,7,12,13,14,15) is incremented by the last element in the previous set of four, giving us [3,4,5,8,8,8]. There is no need to continue to odd items when divided by 8 as the array is smaller than 8. Finally every item is incremented by the current number of elements minus one (to give us IDs), which gives us [8,9,10,13,13,13]. This algorithm is shown in Algorithm 13.

6.2 Colouring and independent sets

In order for correct and consistent parallel execution of these algorithms certain tasks need to be performed in independent sets. This way any modification to the mesh is sufficiently far away from any other modification done concurrently as to not cause any conflicts. Independent sets are produced by colouring the graph. For the CPU implementation a simple first fit colouring technique was used, this is a very serial technique and does not port well onto the GPU. The initial design was to transfer the mesh up to the CPU for colouring and transfer

[2] http://supercomputingblog.com/cuda/cuda-tutorial-3-thread-communication/

Algorithm 13 Reduction style running total algorithm

```
for all items in array do {Launches a parallel thread for each item in the
array, content of this loop is run concurrently}
  blockSize ← 1
  while blockSize < arraySize do
    if itemID & blockSize ≠ 0 then
      array[itemID] += array[itemID - ((itemID & (blockSize - 1)) + 1)]
    end if
    sync threads
    blockSize ← blockSize * 2
  end while
end for
```

it back down again, the hope was that this could be done asynchronously as the whole algorithm does not need these independent sets and by the time the GPU was ready for the these sets they would already have been transfered down. Unfortunately this did not transpire and the GPU was waiting a long time before it could continue. As a result two alternative colouring methods were considered. First we considered partitioning the mesh, colouring the nodes that made up this partition, subsequently colouring the nodes inside each partition in parallel. This is a two level colouring as surveying in the background section. The issues with this approach is the initial partitioning of the mesh, the most efficient way is to use a quad tree, but even this is very difficult on the GPU and it is quicker to perform this on the CPU and then transfer down into GPU memory. The second approach we considered is using the Jones-Plassmann algorithm [JP95], described in the background. This can easily be implemented solely on the GPU and far out performs a two level colouring. Furthermore this algorithm has been experimentally tested for parallel execution [ABC+93]. Although the problem of colouring was not originally in the scope of the project, a complete GPU colouring implementation was created using Jones-Plassman algorithm.

6.3 Pseudo random number generation

In order to implement the colouring discussed in the previous chapter we need to generate random numbers. The purpose of these random numbers is to give an unordered (i.e. the order does not follow the geometric order of the elements in any way) ranking of all the elements, it also must have the property of not repeating the same number. In W. B. Langdon's paper an approach fast pseudo random number generation on CUDA is presented along with an experimental analysis [Lan09]. This is a CUDA implementation of Park and Miller's pseudo random number generator which not only produces a seemingly random sequence of integers it uniformly samples the first 2^{31} - 2 integers without repetition [PM88]. This fulfilled the requirements exactly and was therefore adopted.

The whole implementation is now completely on the GPU, data only needed to be transfered from the host at the start of execution and then transfered back up again.

6.4 Maintaining adjacency information

The initial plan was to neglect the validity of adjacency information (the only adjacency information required is element adjacency) during coarsening and refinement operations and then recreate this information in much the same was as it was initially created before the next coarsening or refinement operation. The adjacency information was also created on the CPU as it only needed to be executed prior to colouring (which as explained above was also initially on the CPU). The performance implications of this were not fully realised until they were tested. The re-creation of adjacency information accounted for 87% (before the code was optimised) of the total execution time. This of course needed to be addressed. To resolve this the adjacency information was instead maintained during coarsening and refinement. Although the implementation of this was rather complicated, it did resolve the performance problems, almost completely eliminating the time for maintaining adjacency information.

6.5 Avoiding vertex-facet adjacency

The usual way to evaluate the validity of an edge collapse and update vertices in an element during coarsening is to maintain a vertex facet adjacency graph. Doing so gives you a list over every facet connected to each vertex. When an edge has been marked for collapse, we mark one vertex for removal and the other vertex, or target vertex, on that edge as the new vertex for any element which will not be removed in the collapse to replace the removed vertex in elements. The list of facets connected to a vertex is traversed, evaluating whether the collapse is valid at each, if so then each facet replaces the vertex marked for removal with the target vertex. The issue with having this information is not only the overheads associated with its creation, storage, transfer to and from GPU memory and maintenance but also the fact that each vertex is connected to an undetermined and unbound number of facets. To effectively implement this in CUDA a bound would have to be set as you cannot have dynamic data structures in CUDA, so an array of sufficient size would have to be used. This will therefore limiting the flexibility of the application as it would prevent certain coarsening and refinement operations [3]. It would also lead to a lot of wasted space as the majority of vertices will not be connected to as many facets as has been allowed for.

To combat this a novel approach was taken. A GPU does not have the same overhead for launching new threads as a CPU, in fact they are designed to

[3] as discussed in the adaptivity algorithms chapter, an edge collapse will increase the degree of vertices, and so will bisection

launch thousands of threads. This fact was exploited by performing the validity checking of an edge collapse for every element in the mesh concurrently. In parallel every facet refers to an array containing information on which vertex (if any) will replace a vertex, it does this for every vertex the element contains. Using these new vertices it performs a validity check and removes the collapse operation if it fails. Similarly every element is updated with the new vertex information if the collapse was not removed in the previous step.

This avoided the need to maintain a potentially expensive adjacency graph and demonstrates how GPU architecture can be exploited. This approach would not be efficient on the CPU as it involves launching hundreds of thousands of threads for even a medium sized mesh. The overheads associated with launching a new thread on a CPU would greatly degrade performance. A similar style could however be adopted using OpenMP which would re-use threads and therefore not suffer so much from the launching of new threads.

6.6 Shared new vertices

Due to the fact that refinement is parallelised on a thread per facet level, if a facet is regularly refined, then each neighbouring facet is also refined (bisected or regularly refined). A new vertex will be created along the edge connected the two neighbouring facets. Both of these facets will need this vertex in creating their children. One way to solve this problem is for every facet to create a new vertex every time it is needed, and at the end of the refinement process all the duplicate vertices are removed. This is extremely inefficient both in terms of computation and memory usage. A better approach is to preform refinement in independent sets, so that two adjacent facets are not refining at the same time, the first facet to execute will inform the adjacent facet to the ID of the vertex it has created along the bounding edge.

The actual approach taken was an adaptation of this. After it has been calculated which edges of a facet will require a new vertex, in independent sets each facet marks the internal ID of the new vertices (i.e. 1,2,3), which later will be put together with all other facets to become the new unique vertex IDs. Each facet will then mark the facet adjacent to the new vertices with their own ID (the ID of the facet not the vertex as the vertex is only unique within that facet). If a facet has been marked with another facets ID, it will not create a new vertex, instead it will allow the other facet to create the vertex and obtain a reference to the newly created vertex later in the refinement step.

This approach means that only a small graph (Facet adjacency graph) needs to be colour. Another major advantage is that only the marking of new vertices needs to be done in independent sets, the remainder of the algorithm can be done completely in parallel.

6.7 CUDA kernels

Below is a selection of information describing the different CUDA kernels which made up the GPU coarsening (Table 6.1) and refinement (Table 6.2) functions. The number of threads launched is denoted in terms of number of facets (prior to any modification), number of new facets created, number of vertices (prior to any modification) and number of colours (in the case of 2D this will always be 4). In coarsening, note that kernels marked with a "*" are called every time the main coarsening loop is executed (see Design chapter), other kernels are only called once.

Kernel Name	Description	Threads per launch
addAdjacentFacetToVertex*	Marks every vertex with a single adjacent element.	1 * Facets
evaluateVertices*	Evaluates every edge connected to a vertex. From the initial adjacent facet it find the adjacent facet which is also connected to the vertex being tested. If it reaches a boundary element it terminates and marks the vertex to not be coarsened.	1 * Vertices
removeAdjacentCollapses*	Defers collapses that would cause conflicts if collapsed concurrently.	1 * Facets
removeInvalidCollapses*	Removes collapses that would either invert and element or create an edge greater than the upper bound.	1 * Facets
collapseFacets*	Performs the collapse.	1 * Facets
updateFacetVertices*	Replaces the removed vertices with the target vertices.	1 * Facets
updateFacetAdjacency*	Updates facet adjacency.	1 * Facets
propagateFacetUpdate*	Propagates updates of facet adjacency in the case where two adjacent facets were collapsed (adjacent facets can be collapsed if they are not collapsing adjacent edges).	1 * Facets
reindexFacets	Calculates new facet IDs when collapsed facets have been removed.	1 * Facets
reindexVertices	Calculates new vertex IDs when collapsed vertices have been removed.	1 * Vertices
compactFacets	Copies facets not removed to a new space, updating references to vertices and adjacent facets.	1 * Facets
compactVertices	Copies vertices not removed to a new space.	1 * Vertices

Table 6.1: CUDA kernels used for coarsening

Kernel Name	Description	Threads per launch
pseudoRandomMarkFacets	Assigns a pseudo random number of every facet.	1 * Facets
colourFacets	Colours the facets using Jones-Plassman colouring. This is repeatedly called until all facets are coloured, this usually takes between 10 and 14 invocations on a large mesh.	m * Facets
evaluateFacets	Every facet is evaluated against the metric tensor, it is marked for refinement if it is not within the user set maximum bound.	1 * Facets
markFacets	This propagates the markings of the facets, marking which elements need to be regularly refined and which need to be bisected. This is called repeatedly until propagation terminates.	n * Facets
memoryIndex	Calculates the number of new facets that need to be created, also assigns a unique ID to each of these new facets.	1 * Facets
markVertices	Called for each colour in turn, facets are marked as to whether they require a new vertex to be created or they will be referencing a vertex owned by an adjacent facet.	Colours * Facets
indexVertices	Calculates the number of new vertices that need to be created, also assigns a unique ID to each of these new vertices.	1 * Facets
populateVertices	Populates the co-ordinates of the newly created vertices.	1 * Facets
dereferenceVertexIndices	Retrieves the vertex ID of vertices referenced from adjacent facets.	1 * Facets
populateFacets	New facets are created.	1 * Facets
updateFacetAdjacency	Facet adjacency is recalculated.	1 * Facets + New Facets

Table 6.2: CUDA kernels used for refinement

6.8 Summary

In this chapter the challenges and complications faced with implemented the application have been discussed. A suitable solution has been found for all of these problems resulting in an application that can be analysed in depth, optimised and finally evaluated, all of these are presented in the next chapter.

Chapter 7

Evaluation and improvements

This chapter describes how the project was tested and evaluated as well as any optimisations made. Firstly the testing environment and method of analysis is described, then an explanation of how any certain improvements were made finally an analysis of the final work and a comparison against a CPU implementation.

7.1 Testing environment

All tests were carried out on machines with identical hardware. These machines were managed by a portable batch system (PBS) queue system [1]. The use of PBS meant that it was assured that only one job was running on the machine at any one time, this allowed us to achieve fair results during testing. Below is the detailed specification of the testing environment (Table 7.1).

[1] http://linux.die.net/man/1/pbs

CPUs	2 x Intel(R) Xeon(R) CPU X5650 @ 2.67GHz
Total number of CPU cores	12 (24 with hyper threading)
CPU core	Westmere-EP
CPUID	206C2h
CPU bus speed	3200MHz
CPU maximum thermal design power	95W
CPU memory	24GB
CPU memory type	DDR3-1333
CPU max memory bandwidth	32GB/s
Cache size	12288 KB
QPI speed	6.40 GT/s
Operating system	Linux 2.6.18-238.5.1.el5
PCIe memory bandwidth	5.3GB/s
Graphics card	Tesla M2050 Fermi
CUDA cores	448
Graphics Peak single precision floating point performance	1030 Gigaflops
Graphics memory size	3GB
Graphics memory type	GDDR5
Graphics memory bandwidth (ECC off)	148 GB/s
GCC Version	4.1.2
GCC compiler flags	-O3
CUDA SDK	3.1
CUDA compilation tools	release 3.1, V0.2.1221
CUDA compiler flags	-O2
nVIDIA Forceware driver	260.19.29

Table 7.1: Testing Environment

7.2 Timings

The work was timed using the CUDA event management module in the CUDA Runtime API [2]. The mesh is initially read in from a vtu file and the classes required for the mesh is initially created, this is not included in the timings as it is common to both GPU and CPU and does not relate to the problem of mesh adaptivity. For the same reason the writing of the output mesh to disk is also not included. Transfer time associated with copying the mesh to and from the GPU memory is included. If not otherwise stated, the times or performance increase figures were obtained by averaging 6 or more runs, 3 with error bounds setup so that the mesh requires a lot of coarsening and 3 such

[2] http://www.clear.rice.edu/comp422/resources/cuda/html/group__CUDART.html

that the mesh requires one refinement of every element. The timings between section which at first might appear to be inconsistent are due to the overheads of profiling and timings. Profiling was turned off whenever possible, and timings not required for a particular test were also turned off. All timings taken in a particular experiment were taken using exactly the same setup and therefore the comparisons are justified.

7.3 Execution time breakdown

The below timings were achieved using an initial mesh of 1,000,000 vertices and 2,000,000 facets which is a fairly large mesh. A range of timings was taken with different error bounds. With low error bounds little coarsening needs to be done, but a large amount of refinement, similarly with high error bounds a lot of coarsening has to be performed and little refining. The timings below show times for a single round of refinement or coarsening (Table 7.2). Accuracy has been reduced to 3 decimal places for readability, more accurate timings will be used when necessary. The minimum time is achieved where no mesh adaptivity has to be performed, this therefore represents the time to determine if any adaptivity needs to be performed. The average time is a median average time. The maximum time is the worst time achieved over the range of error bounds tested.

Event	Minimum	Average	Maximum
Copy mesh to GPU	0.176	0.176	0.176
Coarsening	0.142	120.535	121.047
Prepare mesh for refining	0.001	0.001	0.001
Refinement	0.001	0.344	0.723
Copy mesh to CPU	0.194	0.195	0.611
Total	1.308	121.078	121.080

Table 7.2: Application timings in seconds on initial mesh with 2,000,000 facets

7.4 Improving Coarsening

It is obvious from the above timings that there is a performance issue with coarsening. Looking at coarsening on slightly smaller meshes and this problem is not experienced. With smaller meshes (1,000,000 facets and fewer) the times for coarsening are similar to those of refining (Table 7.3).

Event	Minimum	Average	Maximum
Copy mesh to GPU	0.073	0.074	0.074
Coarsening	0.075	0.075	0.880
Prepare mesh for refining	0.001	0.001	0.001
Refinement	0.020	0.151	0.274
Copy mesh to CPU	0.020	0.246	0.246
Total	0.547	0.547	0.995

Table 7.3: Application timings in seconds on initial mesh with 800,000 facets

The coarsening step works by continually collapsing small edged elements until all small edged elements that would not invalidated the mesh if collapsed. To avoid data conflicts adjacent collapses are deferred until such a time where their neighbours have been collapsed. To achieve this, unlike refining where elements are coloured and then each colour is handled, the exclusion of neighbouring collapses has to be done on the fly as the adjacent elements changes every round of collapses. After elements have been marked for collapse, if a collapse is adjacent to another collapse, it is differed if it is collapsing an long edge or if their edges are equal it is deferred if its vertex marked for removal has a lower ID. It is in this procedure where the problem lies. In theory each iteration may only collapse a single edge, which would therefore lead to an extraordinarily long execution time. This will happen if the vertex id were sequentially increasing.

To test this hypothesis the number of times the coarsening procedure iterated was recorded. For the smaller mesh (800,000 elements) the loop was executed 22 times, whereas for the longer running times in the larger mesh (2,000,000 elements) it was executed 978 times. This was strong evidence to support the hypothesis.

The first thing done to tackle this problem was not to simply defer the collapse with a lower ID, but to defer the collapse with a lower hashed ID. The hope being that this will prevent sequentially increasing IDs from creating an extraordinarily long execution time. This reduced the number of iterations from 978 to just 8 and the coarsening time from 120 seconds to 1.09 seconds.

7.5 Pinned memory

Page-locked or pinned memory is memory that cannot be swapped out to secondary storage by the operating system. If pinned memory is used by the host, then the data transfered to and from the graphics card does not have to go through the CPU and transfers can attain the highest bandwidth between host and device The data is transfered by the graphics card using direct memory access (DMA). On PCIe 16 Gen2 cards, for example, pinned memory can attain greater than 5 GBps transfer rates [3]. Data transfer to and from the GPU

[3] http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf

accounts for 30%-50% of the total execution time on a mesh with 2,000,000 elements. For this reason the data transfered to and from the GPU was pinned on the host side [4]. The results of using pinned memory are shown in Table 7.4 and Table 7.5.

Event	Paged memory time (seconds)	% of total execution time	Pinned memory time (seconds)	% of total execution time
Copy mesh to GPU	0.1770	10.904%	0.1747	7.619%
Copy mesh to CPU	0.6120	37.697%	1.2747	55.575%
Total data transfer	0.789	48.601%	1.4494	63.194%

Table 7.4: Results of using pinned verse paged memory on the host with a mesh of 2,000,000 elements adapted to 8,000,000 elements

Event	Paged memory time (seconds)	% of total execution time	Pinned memory time (seconds)	% of total execution time
Copy mesh to GPU	0.1770	11.998%	0.1747	8.288%
Copy mesh to CPU	0.1324	8.972%	0.2689	12.757%
Total data transfer	0.3094	20.970%	0.4436	21.045%

Table 7.5: Results of using pinned verse paged memory on the host with a mesh of 2,000,000 elements adapted to 1,600,000 elements

Looking at the experiment results the use of pinned memory has actually degraded performance. Time for copying the mesh to the GPU is pretty much unchanged, but time for copying the mesh back to the CPU has increased significantly with the use of pinned memory. This can be explained by looking at the system specification. The maximum memory bandwidth for this type of machine is 32GB/s, much higher than previous generations architectures. This means that the limiting fact is the PCIe bandwidth (5.3GB/s) and not the memory bandwidth as it would be in other systems. This explains why no improvement was achieved, but it does not explain the loss of performance when transferring

[4]It is important to be careful when using pinned memory as the host cannot swap this memory out. You therefore need to leave enough system memory for the rest of the application along with the opperating system.

data back. Allocating pinned memory has certain overheads associated with it as it needs to do a fair amount of reorganisation to ensure the data will not be reallocated as well as applying multi-threading locks to avoid race conditions [5]. These overheads are not observed when transferring data to the GPU as the allocation of memory for the initial mesh is not recorded as it is part of reading in the mesh from file and is common with the CPU implementation.

7.6 Comparison of memory allocation

After investigating the use of pinned memory the issue of memory allocation was explored. The first step was to break down the time time taken for each process involved in transferring data to and from the GPU. Transferring data to the GPU is simply a matter of allocating the memory on the GPU, then copying the data down. Transferring data back to the CPU means first deleting the old mesh, allocating space for the new mesh and then transferring the data to the CPU. Transfer time break down shown in Table 7.6. These were obtained using a heavily refined large mesh (initially 2,000,000 elements, 8,000,000 after adaptation).

Event	time(seconds)	% of total transfer time
Allocate space on GPU	0.01815	2.31%
Transfer data to GPU	0.15783	20.07%
Create GPU only data structures	0.00000	0.00%
Free initial mesh from CPU	0.00682	0.87%
Allocate space for new mesh on CPU	0.00001	0.00%
Transfer data to CPU	0.58301	74.15%
Free GPU memory	0.02046	2.60%
Total transfer	0.78628	100.00%

Table 7.6: Break down of transfer time with a mesh of 2,000,000 elements adapted to 8,000,000 elements

The item with the greatest impact on the total transfer time are the actual transfer of the data, which takes up almost 95% of the total transfer time, rather than time spent allocating and freeing memory. However, because the total transfer time impacts on the application greatly, improving 5% of this time is still worth with. For this we looked at different ways to allocate and free memory. The original implementation uses C-style malloc and free, we also

[5] http://www.cs.virginia.edu/~mwb7w/cuda_support/memory_management_overhead.html

looked at C++-style new and delete; malloc, realloc and free as well as pinned memory for completeness. The breakdown of these timings are in Table 7.7.

Event	C malloc and free time(seconds)	C++ new and delete time(seconds)	C malloc, realloc and free time(seconds)	Pinned memory time(seconds)
Free initial mesh from CPU	0.00682	0.00712	N/A	0.06600
Allocate space for new mesh on CPU	0.00001	0.28778	0.00689	0.47351
Total alloc/free	0.00683	0.2949	0.00689	0.53951

Table 7.7: Break down of transfer time with a mesh of 2,000,000 elements adapted to 8,000,000 elements using different memory allocation techniques

These results show that the appropriate choice for memory allocation is either C-style malloc and free, or C-style malloc, realloc and free, the difference between these to is too low to differentiate accurately. The reason why the C++ new and free are so slow in comparison is because in addition to memory allocation type checking is also performed, the objects constructor and destructor are also called for each item in the array.

7.7 Asynchronous memory copy

One useful feature of CUDA is the ability to execute kernels at the same time as performing memory operations such as transferring data to and from the CPU and allocating new space. If we were able to overlap some of the kernel execution with memory transfers the time for these memory transfers could in part be hidden from impacting overall performance. Unfortunately this cannot be attempted in this application. The first kernel to be executed iterates over every facet in the mesh (facets account for about 69% of the data transfered to the GPU), this is executed in just 6 milliseconds on a large mesh, the next kernel to be executed needs the vertex information (accounts for the remaining 31% of data transfered to the GPU). Therefore a maximum of 6 milliseconds can be hidden using asynchronous memory transfer to the GPU (ignoring any overheads associated with asynchronous transfer). Looking at the transfer back to the CPU, the algorithm terminates when it has determined the mesh lies within the bounds. As soon as this has been determined the application has no more work to do but transfer the mesh back up to the CPU, therefore asynchronous memory transfer cannot be utilised.

7.8 Occupancy

The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. Each multiprocessor on the device has a set of N registers available for use by CUDA thread programs. These registers are a shared resource that are allocated among the thread blocks executing on a multiprocessor. The CUDA compiler attempts to minimize register usage to maximize the number of thread blocks that can be active in the machine simultaneously. If a program tries to launch a kernel for which the registers used per thread times the thread block size is greater than N, the launch will fail. The Tesla M2050 has 32K 32bit registers per multiprocessor. Maximizing the occupancy can help to cover latency during global memory loads. The occupancy is determined by the amount of shared memory and registers used by each thread block. To maximise the occupancy the size of thread blocks needs to be adjusted [6].

To determine optimal occupancy the CUDA occupancy calculator was used to analyse each kernel [7]. The occupancy calculator uses the number of registered required for the kernel as well as the amount of shared memory used. Originally all kernels used 32 threads per block. Table 7.8 and Table 7.9 shows the newly calculated optimum block size.

Kernel Name	Used registers	Bytes of share memory used	Optimum occupancy (number of threads per block)
addAdjacentFacetToVertex	14	0	256
evaluateVertices	48	0	96
removeAdjacentCollapses	25	0	400
removeInvalidCollapses	35	0	128
collapseFacets	12	0	256
updateFacetVertices	14	0	256
updateFacetAdjacency	17	0	256
propagateFacetUpdate	14	0	256
reindexFacets	14	4	256
reindexVertices	12	4	256
compactFacets	18	0	256
compactVertices	16	0	256

Table 7.8: Optimum block size to maximise occupancy for coarsening

[6] http://forums.nvidia.com/index.php?showtopic=31279

[7] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Kernel Name	Used registers	Bytes of share memory used	Optimum occupancy (number of threads per block)
pseudoRandomMarkFacets	16	0	256
colourFacets	16	0	256
evaluateFacets	50	0	128
markFacets	12	0	256
memoryIndex	12	4	256
markVertices	13	0	256
indexVertices	16	4	256
populateVertices	21	0	272
dereferenceVertexIndices	15	0	256
populateFacets	21	0	272
updateFacetAdjacency	17	0	256

Table 7.9: Optimum block size to maximise occupancy for refining

Increasing occupancy will not necessarily improve performance. Performance will only be improved if the kernel is bandwidth bound. If it is bound by computation and not global memory accesses then increasing occupancy may have no effect. To determine if improving the occupancy improves performance accurate timings for each kernel were obtained using the CUDA SDK profiler. The profiler records kernel execution time in microseconds. Table 7.10 and Table 7.11 shows the effect of changing the block size from 32 to the the block size calculated by the occupancy calculator on a large mesh (2,000,000 elements).

Kernel Name	Original execution time(microseconds)	New execution time(microseconds)	speedup
addAdjacentFacetToVertex	34,976	29,792	1.174
evaluateVertices	163,976	143,497	1.143
removeAdjacentCollapses	64,138	56,494	1.135
removeInvalidCollapses	36,324	31,104	1.167
collapseFacets	18,093	15,839	1.142
updateFacetVertices	48,826	42,493	1.149
updateFacetAdjacency	3,348	3,234	1.035
propagateFacetUpdate	83,105	59,573	1.395
reindexFacets	2,604	2,637	0.987
reindexVertices	664	669	0.993
compactFacets	19,285	19,312	0.999
compactVertices	1301	1301	1.000

Table 7.10: Effect of adjusting block size for coarsening

Kernel Name	Original execution time(microseconds)	New execution time(microseconds)	speedup
pseudoRandomMarkFacets	896	422	2.123
colourFacets	46,054	27,479	1.676
evaluateFacets	7,129	3,489	2.043
markFacets	246	122	2.016
memoryIndex	2,333	1,114	2.094
markVertices	14,459	9,019	1.603
indexVertices	8,496	4,309	1.972
populateVertices	17,929	9,438	1.900
dereferenceVertexIndices	10,623	5,514	1.926
populateFacets	106,569	54,707	1.948
updateFacetAdjacency	153,906	98,293	1.566

Table 7.11: Effect of adjusting block size for refining

Almost all of the kernels benefited from optimising occupancy, especially those used for refining.

7.9 Thread divergence

One of the ways GPU computation differs from CPU computation is that every thread being executed in a warp (32 threads) much either be executing the same instruction or no instruction at all. Branches in the code cause thread divergence if all the threads do not execute the same branch. The nature of coarsening and refinement requires a lot of branching, so there is bound to be a large amount of thread divergence. This divergence is shown in Table 7.12 and Table 7.13. Only kernels with significant divergence are shown.

Kernel Name	Number of divergent branches within a warp.
evaluateVertices	23,055
removeAdjacentCollapses	5,655
removeInvalidCollapses	10,674

Table 7.12: Thread divergence for coarsening on a mesh with 800,000 elements

Kernel Name	Number of divergent branches within a warp.
colourFacets	20,758
markVertices	8,377
dereferenceVertexIndices	8,510
updateFacetAdjacency	54,668

Table 7.13: Thread divergence for refining on a mesh with 800,000 elements

Many of these divergences can be explained when looking at the purpose of the kernel. It is also interesting to note that the longest running kernels have the greatest thread divergence, this is strong evidence to suggest that thread divergence is the major limiting factor in performance. First the biggest offended, updating facet adjacency was more closely looked at in terms of thread divergence. This kernel works by first checking that the adjacent facets are correct, and if not it looks up what the correct adjacent facet should be and then updates the current facet. The divergence arises because not every facet will have incorrect adjacency, and it is impossible to group warps such that those which require updating are executed together. One improvement however was made. When correcting an adjacent facet original this was done by passing to a function which takes as arguments the old facet ID and the new facet ID, it then checks each of it's adjacent facets for the one matching the old facet ID and replaces it with the new facet ID. This function was replace by one what takes the new facet ID as well as the position to put it in thus removing the need to check each adjacent facet. This reduced the thread divergence from 54,668 divergence branches within a warp to 43,072, a reduction of 21.2%. This improved performance of this kernel by just over 4%. Analysis of the other kernels did not yield any change.

7.10 Coalesced memory access

An important performance consideration is coalescing global memory accesses [8]. Global memory loads and stores by threads of a warp (half-warp for compute capability 1.x devices) are coalesced by the device into as few as one transaction. In order to achieve coalesced memory access in compute 1.0 and 1.1 devices the k-th thread in a half warp must access the k-th word in a segment, however not all threads need to participate. Compute 1.2 and 1.3 devices improved upon this by coalescing any pattern of access that fits into a segment. This meant that you did not need to be as careful when aligning global memory accesses. On devices of compute capability 2.x, memory accesses by the threads of a warp are coalesced into the minimum number of L1-cache-line-sized aligned transactions necessary to satisfy all threads [9]. Since we are using a compute 2.0 device it is not possible to investigate to what degree global memory accesses are being coalesces as the CUDA profiler does not monitor this for newer devices. Care was taken during implementation for near by threads to access data close to each other.

7.11 L1 cache

For devices of compute capability 2.x the same memory is used for both shared memory and L1 cache. The maximum amount of shared memory used by any of

[8] http://www.scribd.com/doc/49661618/30/Coalesced-Access-to-Global-Memory

[9] http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

the kernels for coarsening and refinement is 4bytes, therefore it makes sense to use more of this memory for L1 cache to improve caching of global memory which will better exploit data locality with more space. The default configuration is 48KB of shared memory and 16KB of L1 cache. This can be changed to 16KB of shared memory and 48KB of L1 cache. This improvement improved the performance of most kernel and gave a performance improvement of 36.67% for coarsening 14.88% for refinement, and an overall performance improvement of 10.88% when tested with a large mesh initially of 2,000,000 elements.

7.12 Final breakdown of execution times

Presented here is a final break down of the execution times after all optimisations have been done. These times were taken by start with a large mesh of 2,000,000 elements which was adapted to an range of different error bounds, an average was then taken (Table 7.14).

Event	Average time (seconds)	% of execution time
Copy mesh to GPU	0.065034	9.91%
Coarsening	0.196306	29.92%
Prepare mesh for refining	0.001006	0.15%
Refinement	0.153593	23.41%
Copy mesh to CPU	0.240145	36.60%
Total	0.656202	100.00%

Table 7.14: Final breakdown of execution times on a mesh with 2,000,000 elements

7.13 Adaptation convergence

Adaptation convergence is how quickly adaptation leads to a mesh that is within error bounds. The number of times the application executes the main adaptivity loop is completely dependent on the metric tensor, error bounds and initial mesh. Every coarsening step will remove every element it can that is below the error bound in a single step, it will also never create an element what is above the error bound. Refinement on the other hand may created elements that are below the error bound when addressing elements above the error bound. If the largest edge in the mesh is N times larger than the upper bound it will require $\lceil log_2 N \rceil$ refinement steps to bring it within the error bounds. As coarsening will not put the mesh outside of error bounds, and it corrects every element put outside error bounds by refinement in a single step the adaption will iterate $\lceil log_2 N + 1 \rceil$ times, the last iteration no adaptivity is performed, the mesh is just evaluated.

7.14 Scalability

Scalability is one of the main design objectives discussed in the design chapter. To experimentally show the application is suitably scalable a range of different initial mesh sizes were tested. These were tested using a range or error bounds, from error bounds that require the mesh to be heavily coarsened to error bounds that require at least two invocations of refinement. These were then averaged to give the results in Table 7.15 and shown graphically in Figure 7.1 and Figure 7.2.

Mesh size (number of facets)	Average time (milliseconds)	Average time per facet (microseconds)
6	1.6412	273.546
36	1.6682	46.340
140	2.1401	15.287
12,432	5.4395	0.437
50,142	13.4411	0.268
112,554	30.0362	0.266
313,344	91.2122	0.291
802,496	243.5656	0.303
2,000,000	361.7561	0.180

Table 7.15: Range of different sized meshes tested

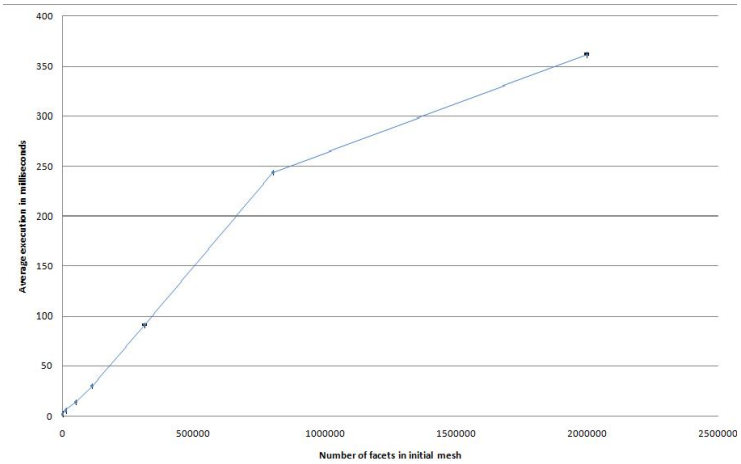

Figure 7.1: Average execution time for different sized starting meshes

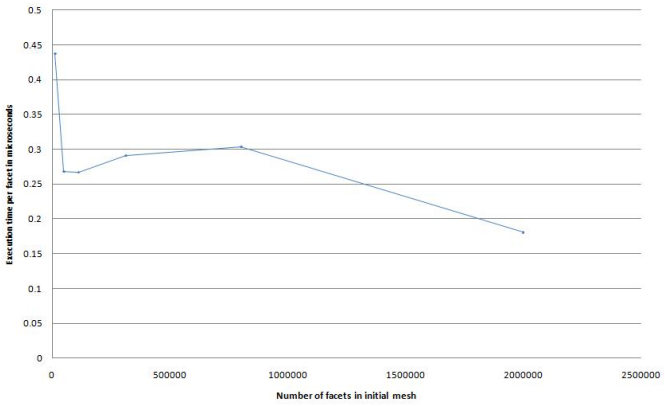

Figure 7.2: Average execution time per facet for different sized starting meshes

From the evidence presented above the application is certainly scalable. The design objective of scalability has been achieved to great effect.

7.15 Comparison with CPU

The two other design objectives are to implement a CPU version of the application for comparison against the GPU version and achieve a good speedup of GPU computation over the CPU computation. A multi-threaded CPU implementation was developed using OpenMP, however during the course of this project work done by Gerard Gorman on an adaptivity suite which included coarsening and refinement was completed. Comparison with CPU was done using this the coarsening and refinement elements of this suite.

To compare GPU and CPU implementations coarsening and refining were analysed separately. The coarsening time represents a single round of heavy coarsening, including the transfer times associated with transferring the mesh to and from the GPU. The refinement time represents a single round of heavy refinement, including the transfer times associated with transferring the mesh to and from the GPU. This is shown in Table 7.16 and Table 7.17.

Mesh size (number of facets)	CPU Average time (milliseconds)	GPU Average time (milliseconds)	GPU Speedup
140	1.9	2.8	0.67
12,432	144.8	9.3	15.54
50,142	677.7	17.6	38.58
112,554	1,569.0	37.7	41.59
313,344	4,531.1	116.1	39.04

Table 7.16: Range of different sized meshes tested against CPU and GPU implementations for a single coarsening round

Mesh size (number of facets)	CPU Average time (milliseconds)	GPU Average time (milliseconds)	GPU Speedup
140	3.7	4.2	0.87
12,432	470.2	7.6	62.15
50,142	3,799.8	20.9	181.94
112,554	6,490.4	46.8	138.70
313,344	7,765.5	137.0	56.67

Table 7.17: Range of different sized meshes tested against CPU and GPU implementations for a single refinement round

For very small meshes the GPU is slightly slower than the CPU, but these are trivial cases where adaptivity is under 5ms. With coarsening, the GPU is about 40 times faster than the CPU for meshes with more than 50,000 facets. The refinement case is interested, the speedup peaks for medium sized meshes (50,000 elements), this is likely to be due to the overheads associated with the launching of new threads, for the smaller meshes it is unlikely that any multithreading is needed, so this overhead is not observed. The speed settles at about 55 times speedup for large meshes.

It is important to note that the tested case isolates a single round of coarsening or refinement, this this being the case the overheads of transferring data to and from the GPU are exaggerated, more impressive speedups will be observed when greater mesh adaptation is performed. The comparison was testes this way because it is easy to ensure testing of like for like cases resulting in a fair and valid test.

The speedup achieved for larger meshes is impressive and means that our performance objectives have been achieved. In a real world application timings and difference between CPU and GPU implementation can vary greatly. For meshes that require very little adaptation, and speed up of around 15 times can be expected, whereas for a large mesh which requires 3 refinement steps a speedup in excess of 100 times is expected.

7.16 Limitations

The application is subject to some constraints, this section explains these and actions, if any, that can be taken to overcome them.

7.16.1 Adaptivity limitations

The choice of using isotropic refinement with anisotropic coarsening does bring with it some limitations to what adaptivity can be done. In the case where a badly formed element is surrounded by large elements it cannot be removed by coarsening as it would create element(s) with edge lengths above the upper bound and of course it cannot be removed though refinement. If a particular metric tensor results in this type of scenario often occurring, this application would be ill-suited to solving it. The addition of edge flipping (preformed once per iteration) will help to fix a lot of these cases.

7.16.2 Thread limitations

In the current implementation a 1D grid is used. This limits the number of blocks that can be run to 65,535, this means that the application can launch a maximum of 8,388,480 threads per kernel invocation (6,291,360 threads for the evaluateVertices kernel, but this is done over vertices which is generally half the number of facets). The maximum number of threads launched for a single kernel invocation is one thread per facet, therefore the maximum number of facets is 8,388,480. This limit applies before and after adaptivity has been preformed. This limitation is easy to overcome. A simple change to used 2D grids would solve this, raising the facet limit to 549,739,036,800. In the unlikely event that this is not enough (impossible in fact due to memory limitations stated below) then each kernel invocation could be wrapped in a loop, each iteration working on a segment of the facets, this would completely remove the limit of maximum threads.

7.16.3 Memory limitations

The current environment has 3GB of memory on the graphics card, the whole mesh before and after adaptation needs to be held in this memory, along with other data structures described in the design chapter. Facets occupy 32 bytes each and vertices occupy 40 bytes each, there is usually twice as many facets as vertices, for reasonably sized meshes this is always the case to within a few percent, we can therefore say the mesh consists of 52 bytes per facet. Including all the other data structures and addition 42 bytes per facet is required. When the mesh size is increased or decreased (as it inevitably will be during coarsening and refining) extra space is needed to store the old mesh as it is being coped to the space allocated for the new mesh. New vertices and facets are created at different times, and the temporary space required for the of copying one is freed before the start of copying the other. This therefore adds an extra memory

overhead of 32 bytes per facet. The total memory required on the graphics card is 126 bytes per facet putting a limitation of 23,809,523 facets, this being the maximum facets at any time during the execution of the application. One way around this is to partition the mesh on the CPU and then adapt each partition in turn on the GPU. These partitions will then have to be stitched together. Partitioning a mesh is a non-trivial problem and is beyond the scope of this project.

7.17 Summary

The performance of the application has been thoroughly analysed and evaluated. The performance figures are beyond what was initially expected and are the key to marking this project as a success. The following chapter will look at where to go next in continuing this research, as well some final thoughts and reflections.

Chapter 8

Conclusion

This chapter will conclude the report and look at interesting areas for future work. A brief section is devoted to reflecting on any lessons learned and justifying the success of the project. Finally the report is concluded with some closing remarks.

8.1 Future work

There is a variety of topics which remain open for further study. Below the most significant of these are listed.

- The most immediate item for further work is to integrate this work into a common framework for GPU mesh adaptivity where it can be coupled with work already completed on smoothening. This will provide a platform for addition mesh adaptivity algorithms on the GPU such as edge flipping and other coarsening and refinement techniques.
- Edge flipping is a mesh adaptivity technique that is the logical successor to smoothening, coarsening and refinement. Implementation of edge flipping coupled with the work presented in this thesis will lead to a complete suite of mesh adaptivity.
- A new type of processor combining CPU style cores and GPU style cores on a single chip is being developed by AMD, called an accelerated processing unit (APU) [Bro10]. With the GPU core and CPU core on a single chip, as well as access to a common memory the times associated with transferring data to and from the GPU (which accounts for about 50% of the total execution time) can be eleminated. The pure compute capailities of this new chip are unlikely to be as powerful as a GPU, but the elimination of transfer times might mean that performance can still be improved. An APU implementation of this application and comparision against this GPU application would be of great value.

- An obvious extension is to move from 2D to 3D. It is not a simple matter of stating that this work can easily be generalised to the 3 dimensional case and still maintain its performance advantage; some careful thought needs to be taken. Firstly more colours will be required, in 2D only 4 colours are needed as each facet has 3 neighbours. In 3D the mesh would consist of tetrahedrons, so 5 colours will be required, a slight degradation of performance. Calculations such as evaluation of the metric tensor at particular points will be more complicated, as well was calculating whether a proposed collapse would invert an element. The extra computation cost of these will be a factor less than 2. The memory footprint will also increase slightly, as each facet will need to hold 4 vertices instead of 3 and 4 adjacent facets instead of 3. Each vertex will also need to hold an additional co-ordinate. The other data structures will largely remain the same. In 3D 164 bytes per facet will be needed an increase of 30% over 2D. There are also a number of other factors that are extremely dificult to predict their impact such as the inevitable increase in the number of register required for many of the kernels.

8.2 Reflection

Looking at the project in its entirety several reasons for its success become apparent. The main reason for the success is the strategic approach taken to the problem, the extensive background research completed, and the benefit of building on Gerard Gorman's experience of implementing mesh adaptivity on multi-core systems.

This project can also benefited from the use of a very large number of threads, several million threads per kernel and hundreds of millions per execution on a large mesh. GPUs do not suffer from the same overheads associated with creating new thread as CPUs, good performance can only be achieved on a GPU if the device is utilised by providing it with enough threads.

Another reason why this project is successful is how the procedures for coarsening and refinement were broken down into many kernels (12 for coarsening and 11 for refinement) so that consistency is obtained by serial calling of these massively parallel kernels. The decomposition of the adaptivity algorithms to yield these different kernels was one of the greatest challenges of this project and many novel techniques were employed in achieving it. This decomposition has an added effect of limiting the amount of work that has to be done in independent sets. In previous work done on parallel mesh adaptivity the vast majority of work was done on independent sets, whereas specifically for refinement, only a very small proportion of this work done required the use of independent sets.

The intelligent choice of data structures allowed for smaller graphs of dependent elements, reducing the total number of independent sets. The careful selection of data structures also reduced the amount of information that needed to be maintained during mesh adaptivity as well as limiting the memory foot-

print of the application.

Finally the efforts taken beyond the scope of the project to move all computation onto the GPU was vital in getting an high performing end to end solution. In particular moving graph colouring from the CPU to the GPU meant the entire computation is executed solely on the GPU.

8.3 Closing remarks

This project has been highly successful. Not only were the design objectives met and surpassed, but the completed work will also be developed further as part of Georgios Rokos PhD almost immediately after completion [Rok11]. The project has shown that massive speedup can be achieved using GPUs for an application with unstructured data, irregular memory access and complex algorithms. The full extent to which this project has contributed to the field of computational fluid dynamics will not be realised until some of the future work described above has been completed. Great effort is being made to progress techniques for mesh adaptivity on GPUs, and it is highly likely this work will make its way into high performance fluid solvers in the near future. The potential applications of this are extensive, from salt fingering to ocean modeling, the prospective influences it will have on computation fluid dynamics are vast.

Bibliography

[ABC+93] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin. A comparison of parallel graph coloring algorithms. *SIAM Journal of Scientic Computing*, 14:654, 1993.

[ALSS05] Frdric Alauzet, Xiangrong Li, E. Seegyoung Seol, and Mark S. Shephard. Parallel anisotropic 3d mesh adaptation by mesh modification. *Engineering with Computers*, 21(3):247–258, 2005.

[BLS99] Andreas Brandstdt, Van Bang Le, and Jeremy Spinrad. *Graph Classes: A Survey*. Discrete Mathematics and Applications. SIAM, Philadelphia, PA, USA, 1999.

[Bro41] R. L. Brooks. On colouring the nodes of a network. *Mathematical Proceedings of the Cambridge Philosophical Society*, 37:194–197, 1941.

[Bro10] Nathan Brookwood. Amd fusion family of apus: Enabling a superior, immersive pc experience. *Insight*, 64:1–8, 2010.

[BSW93] Randolph E. Bank, Andrew H. Sherman, and Alan Weiser. *Some refinement algorithms and data structures for regular local mesh refinement*, pages 3–17. Scientific Computing. IMACS/North-Holland Publishing Company, Amsterdam, 1993.

[Chv84] Vclav Chvtal. Perfectly orderable graphs. *Topics in Perfect Graphs, Annals of Discrete Mathematics*, 21:63–68, 1984.

[FJP98] Lori A. Freitag, Mark T. Jones, and Paul E Plassmann. The scalability of mesh improvement algorithms. *Institute for Mathematics and Its Applications*, 105:185, 1998.

[JP95] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *Work*, 1995.

[JP97a] Mark T. Jones and Paul E. Plassmann. Adaptive refinement of unstructured finite-element meshes. *Finite Elements in Analysis and Design*, 25:41–60, 1997.

[JP97b] Mark T. Jones and Paul E. Plassmann. Parallel algorithms for adaptive mesh refinement. *SIAM Journal on Scientific Computing*, 18(3), 1997.

[KKT01] Jrg Keller, Christoph Keler, and Jesper Trff. *Practical PRAM Programming*. John Wiley and Sons, 2001.

[KmWH10] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach*. Elsevier Inc, 2010.

[Lan09] W. B. Langdon. A fast high quality pseudo random number generator for nvidia cuda. *GECCO '09 Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, 2009.

[LSB05] Xiangrong Lia, Mark S. Shepharda, and Mark W. Beallb. 3d anisotropic mesh adaptation by mesh modification. *Computer Methods in Applied Mechanics and Engineering*, 194(48-49):4915–4950, 2005.

[McG11] Timothy P McGuiness. Parallel mesh adaptation and graph analysis using graphics processing units. Msc thesis, University of Massachusetts Amherst, February 2011.

[PM88] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 32(10):1192–1201, 1988.

[RAO98] Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi. Stapl: Standard template adaptive parallel library. *LCR*, '98:402–409, 1998.

[Riv84] Maria-Cecilia Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM J. on Numerical Analysis*, 21:604–613, 1984.

[Rok10a] Georgios Rokos. Accelerating optimisation-based anisotropic mesh adaptation using nVIDIAs CUDA architecture. Msc thesis, Imperial College London, September 2010.

[Rok10b] Georgios Rokos. Study of anisotropic mesh adaptivity and its parallel execution. iso-report, 2010.

[Rok11] Georgios Rokos. Strategies for accelerating mesh adaptation for fluid dynamics, June 2011.

[VL99] Y. Vasilevskii and K. Lipnikov. An adaptive algorithm for quasioptimal mesh generation. *Computational Mathematics and Mathematical Physics*, 39:14681486, 1999.

[Wal01] C. Walshaw. A multilevel approach to the graph colouring problem. Technical Report 01/IM/69, School of Computing and Mathematical Science, Univeristy of Greenwich, London, UK, May 2001.

[WP67] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.