Imperial College London
Department of Computing

Accelerating Optimisation-Based Anisotropic Mesh Adaptation using nVIDIA's CUDA Architecture

Georgios Rokos

Submitted in part fulfilment of the requirements for the
MSc Degree in Computing Architecture of Imperial College London,
September 2010

Abstract

This report presents the design and implementation of 2D anisotropic mesh adaptivity on nVIDIA's CUDA architecture. Mesh smoothing is a component of anisotropic mesh adaptivity, a numerical technology of some importance in Computational Fluid Dynamics (CFD). Adapting the size and shape of elements in an unstructured mesh to a specification encoded in a metric tensor field is done by relocating mesh vertices using an optimisation algorithm, like the one proposed by Pain et al. in 2001. This computationally heavy task can be accelerated by engaging nVIDIA's CUDA, a massively parallel and floating-point capable architecture. In order to ensure correct parallel execution, we implemented the parallel framework based on the use of independent sets proposed by Freitag et al. in 1998.

The report contains all related algorithms and architectural details, gives design and implementation descriptions and lists various CUDA code optimisations which can lead to a speedup of up to 190 times over the serial CPU code and up to 45 times over an eight-threaded OpenMP code. Performance analysis shows that CUDA's texture memory can assist in accelerating execution by 2.5 times and that high register usage is the main limiting factor for better performance. The report closes with a short comparison between CUDA and the older Cell Broadband Engine Architecture, explaining why programming in CUDA is easier and expected to yield better performance results.

Acknowledgements

I would like to express my gratefulness and thank following people for their contribution to the accomplishment of this project:

- Professor Paul Kelly, who took the initiative, proposed this project and accepted to take me under his supervision, giving me the chance and all necessary lab equipment to build *CUDAMesh64* and fulfil the requirements for my MSc degree.

- Dr Gerard Gorman, who supported me throughout this project, giving me invaluable advice and guidance, providing all necessary algorithmic knowledge and assuming the role of the second supervisor, devoting a great amount of his time.

- Graham Markall, who assisted me in the technical part of the project, having interesting discussions with me on nVIDIA's CUDA, an architecture with which I had no previous experience, and providing lab support.

- My family who shouldered the financial requirements of my MSc course and provided emotional support against all difficulties of living abroad for the first time in my life.

'I do not think there is any thrill that can go through the human heart like that felt by the inventor as he sees some creation of the brain unfolding to success... Such emotions make a man forget food, sleep, friends, love, everything.'

Nikola Tesla

Contents

Chapter 1

Introduction

Mesh adaptivity is an important numerical technology in *Computational Fluid Dynamics* (CFD). CFD problems are solved numerically using unstructured meshes, which essentially represent the discrete form of the problem. In order for this representation to be accurate and efficient, meshes have to be adapted according to some kind of error estimation. Furthermore, this error estimation may also encode information about a possible special orientation of the problem under consideration, in which case we say that the underlying dynamics is anisotropic.

Various techniques and algorithms can be used to adapt a mesh. One sophisticated technique, which is considered to yield very good results, is the optimisation-based vertex smoothing [FJP95]. It can be implemented using a great variety of algorithms, each one having its own computational advantages and disadvantages over the others, therefore being more or less suitable for a given computer architecture. In this project we only investigated two-dimensional cases, although a large part of the algorithmic background and codebase could be used and extended to three-dimensional problems.

1.1 Motivation and Objectives

Adapting a mesh to an error estimation involves an enormous amount of floating-point operations which can push even the most powerful processing units to their limits. Modern trends like nVIDIA's CUDA and ATI's Stream architectures have made the tremendous floating-point processing capabilities of GPUs available to a wide range of applications apart from their traditional usage in games, graphics and motion picture processing. This shift in GPU target applications has been very successful so far and we are already speaking of supercomputers consisting of graphics processors. Moreover, there is a dormant tendency in CPU industry to slowly integrate the two traditional processing worlds: general purpose (CPU) and massive floating-point (GPU) computing. There are already some early attempts towards this direction, like IBM's Cell Broadband Engine Architecture.

The CUDA platform offers great computational power at relatively low cost (in terms of space occupied, power consumption during operation and money spend to acquire the essential hardware). These properties make it a perfect candidate for accelerating mesh adaptation software. The purpose of this project is writing a new application framework which implements various optimisation-based vertex smoothing algorithms along with the proposal by [FJP98] for their parallel execution, enabling mesh adaptation to be accelerated on CUDA GPUs. It is interesting to perform a comparison between this new platform and conventional hardware and see how well CUDA can compete against the old trends in terms of performance and scalability.

1.2 Related Work

During the past years, the Applied Modelling and Computation Group at Imperial College London has developed a CFD code called "Fluidity". Fluidity is a general-purpose numerical solver for fluid dynamics problems which uses the finite element methods with unstructured finite element meshes. So far it has been used in a wide variety of problems such as heat transfer, oceanic flow and other complex, compute-intensive applications. It was designed to run on conventional hardware and take advantage of distributed memory parallel computers using the *Message Passing Interface* (MPI) [ML09]. Complexity of modern CFD applications, however, shows that calculations are expensive; using conventional hardware requires thousands of processors and many days of processing.

Taking this into account, it becomes obvious that it would be useful to the modelling community to have an extensible, CUDA-enabled framework so that the group can actually take advantage of modern, high-performance GPUs in their modelling and simulation tasks. This framework is built with extensibility in mind so that new adaptation algorithms can be easily incorporated in the future.

1.3 CUDA and CFD

CUDA has already been used to create solvers for computational fluid dynamics problems. Relevant works include [GBT06], which was an attempt to enable CUDA capabilities in an existing application (much like the case of Fluidity), [TccS09], which constitutes an exploration into solving the Navier–Stokes equation on desktop systems using single, dual and quad GPU configurations and [ELD08], which addresses the simulation of a hypersonic vehicle configuration.

In all cases, execution performance was impressive compared to conventional architectures. [TccS09] managed to get a speedup of 100 times using a Quad Tesla S870 server, compared to serial CPU code running on an AMD Opteron 8216. A key point in application performance, which also scales well as the number of GPUs increases, proved to be the extensive usage of shared memory. [GBT06] report that CUDA computing does not perform very well on small problems because the whole overhead of engaging the GPU outweighs the few benefits obtained by low-level parallelism. Additionally, if CPU is assigned with global reduction tasks like result accumulation or data updates there is a great amount of time when GPUs remain idle. Another important conclusion coming from this work is that the single-precision restriction on floating-point arithmetic does not seem to affect result accuracy (in this project, though, we work on *Fermi* chipsets which are proportionally powerful in double-precision arithmetic). [ELD08] have derived some interesting results by analysing their application's performance model. They propose using texture memory to merge small data blocks into one large group so that computational load is kept high at all times and use shared memory with the purpose of increasing memory bandwidth for stencil operations.

1.4 Contributions

The venture of accelerating the optimisation-based anisotropic mesh adaptation on CUDA has led to the following contributions:

- This is the first adaptive mesh algorithm implemented on CUDA as far as the author is aware.
- A speedup of 190 times was achieved over a conventional CPU, which is relatively high compared to other codes.

- Using texture memory to store the metric tensor field and exploiting its dedicated hardware to interpolate metric tensor values offered a speedup of 2.5 times over the simple CUDA code.

In order to carry out this research, we built an extensible mesh adaptation framework, which could be the base for future development projects. Using the same framework, it was estimated that the algorithmic complexity of the anisotropic mesh adaptation problem is $\Theta(n^2)$, n being the number of mesh vertices. Finally, the experience gained throughout this project led to some interesting conclusions on the ease of programming and expected performance from nVIDIA's CUDA as it is compared to IBM's Cell Broadband Engine, based on the author's former experience with the latter platform [RPK+10].

1.5 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 acknowledgement 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 project's design and conception or in style, presentation and linguistic expression is acknowledged.

1.6 Report Outline

The rest of the report is organised as follows: Chapter 2 gives a comprehensive description of the main principles and algorithms that govern the topic of *Anisotropic Mesh Adaptivity*. More precisely, Section 2.1 contains a description of the concepts behind anisotropic problems and mesh adaptivity, along with optimisation-based algorithms, and Section 2.2 describes the framework upon which these algorithms can be executed in a parallel architecture. Chapter 3 is an introduction to the basics of the *CUDA* architecture, presenting CUDA's basic architectural points and outlining some key points and strategies used to achieve high performance on this platform. In Chapter 4 we describe the design choices and the actual implementation of the target application, the classes which encapsulate the entities of the adaptivity problem, the data structures used, various code optimisations that have been investigated while trying to improve execution performance and, finally, a list of difficulties we encountered during the development phase of this application. Chapter 5 presents performance diagrams for *CUDAMesh64* running in single-threaded mode, OpenMP-mode and CUDA-mode. Based on output meshes, we also try to estimate the algorithmic complexity of the anisotropic mesh adaptivity problem. Finally, Chapter 6 summarises the main concepts, the important papers, the experience gained throughout this project and lists the topics that remain open for further study and future work.

Chapter 2

Background Theory

This chapter describes the necessary background knowledge the reader should be familiar with before proceeding to the rest of the report. The research is based on two major subjects, the numerical analysis topic of *Anisotropic Mesh Adaptivity* and the general-purpose computing architecture of *nVIDIA's CUDA*, which is described in Chapter 3. Both of them need to be briefly introduced along with references to a collection of documents and publications, which provide more in-depth details on these topics.

2.1 Anisotropic Mesh Adaptivity

2.1.1 Partial Differential Equations and Meshes

Fluid Dynamics is the science that studies problems related to the motion of fluids. As in all other physics problems, a fluid dynamics problem is modelled using mathematical formulations such as differential and integral equations. In particular, we study such problems using a set of PDEs known as *Navier-Stokes Equations*. In most cases, these equations cannot be solved in an analytical way, so we turn to numerical methods. On common approach is known as *Finite Element Method* (FEM), in which the problem space of the equation under consideration is discretised into smaller sub-regions, usually of triangular (in two dimensions) or tetrahedral (in three dimensions) shape. These sub-regions, referred to as *elements* or *facets* form what is called a *mesh*. The equation is then discretised and solved on this mesh, inside each sub-region.

If there is no regular pattern in the topology of elements in a mesh, the mesh is said to be *unstructured*. Unstructured meshes offer greater flexibility in the Finite Element Method, but their representation is more complex and models based on unstructured meshes exhibit higher computational cost compared to structured mesh models. [PFW+09]. An example of space discretisation which results in an unstructured mesh is shown in Figure 2.1 [cL99].

The numerical solution process can be seriously affected by the quality of space discretisation. Common discretisation techniques often provide us with poor quality meshes and this has the side effect of both slowing down convergence speed and degrading solution accuracy [FJP95]. This is the point where mesh improving techniques, which reside in the wider context of *adaptive algorithms*, come to the foreground. After solving the PDE on the initial mesh and making a posteriori estimations about solution errors we can spot problematic areas and perform these optimisation techniques, which decide on the quality of a mesh element using some kind of local quality metric (for example, minimum

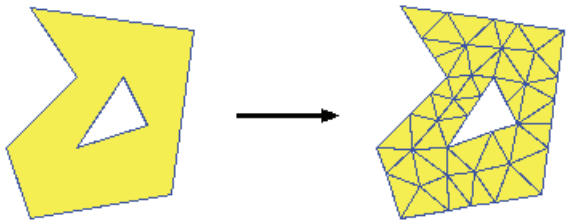

Figure 2.1: Example of space discretisation, resulting in an unstructured mesh. (Figure from [cL99])

element angle) and try to improve element size, angles and ratio; in other words, they try to "adapt" the mesh towards the correct solution.

This process is summarised in Algorithm 1. The first step consists of solving the equation on the initial mesh and spotting the areas where the solution error is greater than a pre-defined tolerance. Then the adaptive algorithm tries to improve mesh quality and, therefore, solution accuracy. This loop is repeated until solution error reaches an acceptable level.

Algorithm 1 General algorithm for the adaptive solution of a PDE.

```
Mesh mesh = new Mesh(PDE);
mesh.adapt();
PDE.solveOn(mesh);
while(PDE.getSolutionErrorEstimation() > USER_DEFINED_THRESHOLD)
{
    Set S = {elements of low quality};
    mesh.adapt();
    PDE.solveOn(mesh);
}
```

2.1.2 Objective Functionals: Element Size and Shape

The quality and, therefore, the suitability of a mesh element is determined by some criterion. Common quality criteria include the size of element angles, the aspect ratio between element edges, element size and element shape. Mathematically, these criteria are expressed in the form of an objective functional, i.e. a functional the value of which corresponds to the "height" of quality.

Vasilevskii & Lipnikov proposed a functional [VL99] which takes both element size and shape into account. The functional is defined as:

$$Q_M(\Delta) = \underbrace{12\sqrt{3}\frac{|\ \Delta\ |_M}{|\ \partial\Delta\ |_M^2}}\ \underbrace{F\left(\frac{|\ \partial\Delta\ |_M}{3}\right)}$$

In the functional above, $|\ \Delta\ |$ denotes the element area of an element Δ and $|\ \partial\Delta\ |$ denotes its perimeter. $F(x)$ is a smooth function defined as

$$F(x) = \left[min\left(x, \frac{1}{x}\right)\left(2 - min\left(x, \frac{1}{x}\right)\right)\right]^3$$

which has a single global maximum of 1 in $x = 1$ and decreases as we move towards 0 or ∞, with $F(0) = F(\infty) = 0$. The first factor expresses the element quality based on its shape, whereas the

second factor controls the size of element edges. For an equilateral triangle with edges of length l, the first factor evaluates to 1. For any other triangle, this factor evaluates to < 1. Additionally, the second factor evaluates to 1 for a triangle with edges the sum of which is equal to 3, e.g. an equilateral triangle with edges of unit length, otherwise it evaluates to < 1. Taking these two factor together, the objective functional reaches its maximum for an equilateral triangle with sides of unit length; in any other case, the objective functional evaluates to < 1.

Usually, and in the scope of this project, we consider the value of the objective functional for the entire mesh or a mesh cavity to be equal to the value of the element with the lowest quality in the mesh or cavity, respectively.

2.1.3 Anisotropic PDEs

A problem is characterised as "anisotropic" if its solution exhibits directional dependencies. In other words, an anisotropic mesh contains elements which have some (suitable) orientation. An example of an anisotropic mesh is shown in Figure 2.2 [cL99]. The process of anisotropic mesh adaptation begins with a (usually automatically) triangulated mesh as input and results in a new mesh, the elements of which have been adapted according to some error estimation. This estimation is given in the form of a *Metric Tensor*. In the scope of this project, we assume that the error estimation in given to us and we do not examine how it can be calculated. An example of gradually adapting a mesh to the requirements of an anisotropic problem is shown in Figure 2.3 [AMC06].

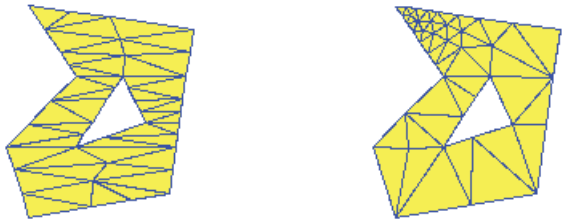

Figure 2.2: Example of an anisotropic mesh. Mesh elements have different shape and size in different locations on the mesh. (figure from [cL99])

The error estimation gives information about how big or small a mesh element should be. In 1-D, the solution error inside an element e (i.e. a line segment) is defined as

$$\varepsilon = h_e^2 \mid \frac{\partial^2 \psi}{\partial x^2} \mid$$

where h_e is the length of element e and ψ is the solution variable. In multi-dimensional problems, the error is defined as

$$\varepsilon = u^T \mid H \mid u$$

where H is the Hessian of the solution equation and u is a vector which shows the ideal length and orientation of element e. In simpler words, the higher the error inside an element e the smaller this element has to become [PUdOG01].

2.1.4 Metric Tensors

In the last equation, the vector u is constructed according to a metric tensor M, i.e. a tensor which, for each point in the 2-D (or 3-D) space, represents the desired length and orientation of an edge

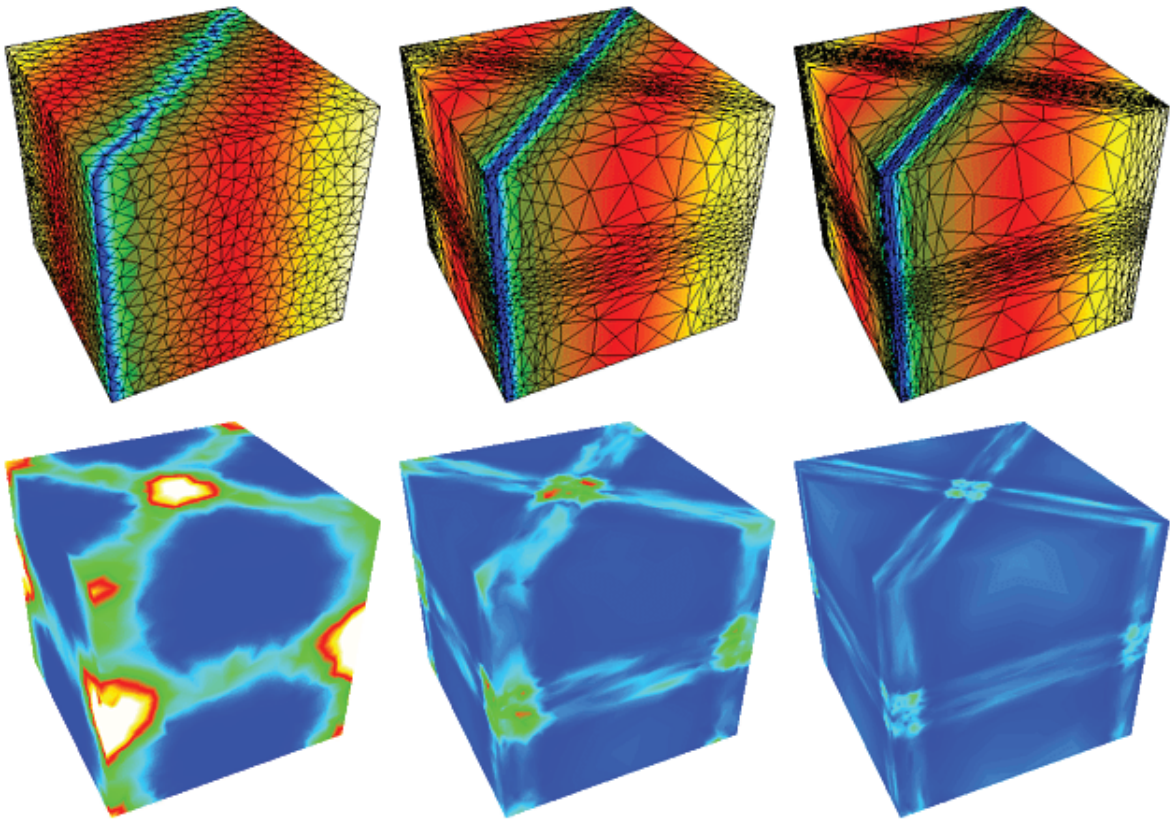

Figure 2.3: Example of anisotropic mesh adaptation. On the left, the process begins with an automatically triangulated mesh, which was not created with with the error estimation in mind. This fact has an impact on the solution error, which is depicted in the bottom left figure. After one refinement iteration (middle figures) the mesh is better adapted to PDE's anisotropy and the solution error is greatly reduced. After two iterations (right figure) the results are even better. (figure from [AMC06])

containing this point. As was the case with the PDE itself, the metric tensor is also discretised; more precisely, it is discretised node-wise. The value of the error between mesh nodes (points) can be taken by interpolating the error from nearby nodes.

The objective functional introduced earlier involved the area and the perimeter of an element E. These quantities are defined with respect to a metric M. If we are in the standard Eucleidian space, the functional reaches its maximum when we have a mesh consisting of uniformly sized, equilateral triangles. In an anisotropic problem we can use the quantities of area and perimeter if we express them with respect to a non-Eucleidian metric $M(\mathbf{x})$. For an element E with area $\mid \Delta \mid_E$ and edges of length $\mathbf{e}_i$ in the standard Eucleidian space, its area with respect to the metric $M(\mathbf{x})$ can be calculated as

$$\mid \Delta \mid_M = \sqrt{\det(M)} \mid \Delta \mid_E$$

and it perimeter as

$$\mid \partial\Delta \mid_M = \sum_{i=1}^{3} \parallel \partial\mathbf{e}_i \parallel_M = \sum_{i=1}^{3} \sqrt{\mathbf{e}_i^T M \mathbf{e}_i}$$

where we consider that M is constant over the element E, both for simplicity and numerical implementation.

Adapting a mesh so that it distributes the error uniformly over the whole mesh is, in essence, equivalent

to constructing a uniform mesh consisting of equilateral triangles with respect to the metric M. This concept can be more easily seized if we give an analogous example with a distorted space like a piece of rubber that has been stretched (see Figure 2.4). In this example, our domain is the piece of rubber and we want to solve a PDE in this domain. According to the objective functional by Lipnikov (2.1.2), all triangles in the distorted (stretched) piece of rubber should be equilateral with edges of unit length. When we release the rubber and let it come back to its original shape, the triangles will look compressed and elongated, a picture that resembles Figure 2.3.

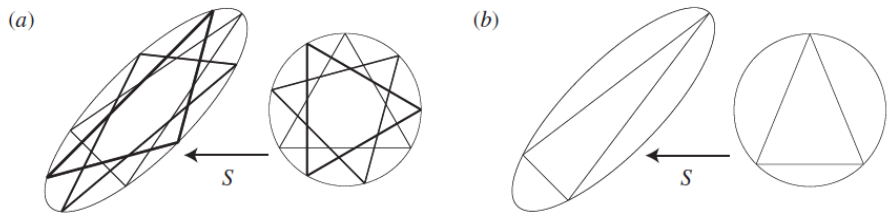

Figure 2.4: Example of mapping of triangles between the standard Eucleidian space (left shapes) and metric space (right shapes). In case (α), the elements in the physical space are of the desired size and shape, so they appear as equilateral triangles with edges of unit length in the metric space. In case (β), the triangle does not have the desired geometrical properties, so it does not map to an equilateral triangle in the metric space.(Figure from [PFW+09])

The metric is defined in such a way that an edge of an element is of unit length with respect to this metric if it has the desired error ϵ_u indicated by this metric, i.e.

$$M = \frac{1}{\epsilon_u \mid \bar{H} \mid}$$

The metric tensor M can be decomposed as

$$M = Q\Lambda Q^T$$

where Λ is the diagonal matrix, the components of which are the eigenvalues of M and Q is an orthonormal matrix consisting of eigenvectors Q^i. Geometrically, Q represents a rotation of the axis system so that the base vectors show the direction to which the element has to be stretched and Λ represents the amount of distortion (stretching). Each eigenvalue λ^i represents the ideal length of an edge in the direction Q^i. If we denote the diagonal values of the Hessian of the solution as h^i, then each eigenvalue λ^i is defined as

$$h^i = \frac{1}{\sqrt{\lambda^i}}$$

so that stretching or compressing an element will be done in an inverse square fashion with respect to the error metric [PFW+09].

2.1.5 Vertex Smoothing

Vertex smoothing is an adaptive algorithm which tries to improve mesh quality by relocating existing mesh vertices. Contrary to other techniques, vertex smoothing has the advantage of leaving the mesh topology intact. All elements affected by this relocation form an area on the mesh called *cavity*. Essentially, a cavity is defined by its central, free vertex and all incident vertices. A vertex smoothing algorithm tries to improve the quality of cavities containing bad elements. Optimisation takes into account only elements belonging to the cavity, which means that only one vertex is considered for relocation at a time. In other words, the scope of optimisation is the cavity, therefore vertex smoothing

is said to be a *local* optimisation technique. The algorithm moves towards the global optimum through a series of local optimisations. An example of optimising a cavity is shown in Figure 2.5 [Rok10].

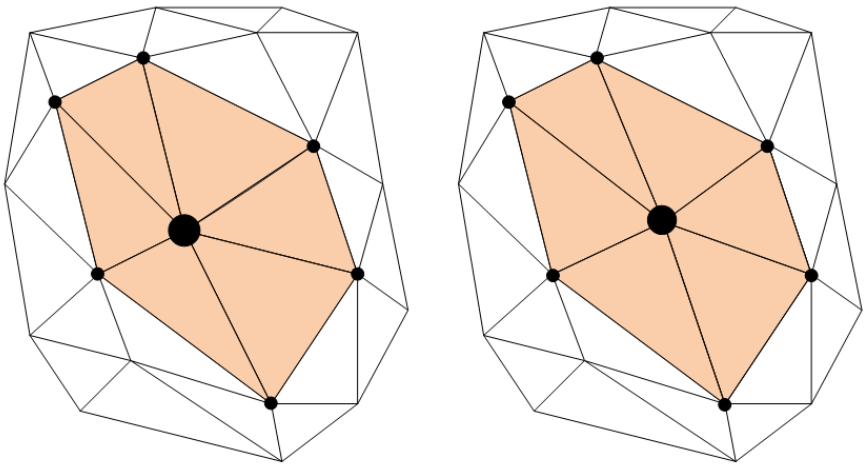

Figure 2.5: Vertex smoothing example. The vertex under consideration is the one marked with a big black circle. The local problem area is the light-orange one. All neighbouring vertices are denoted with small black circles. Left figure shows the local problem cavity before smoothing. Right figure shows the result of local smoothing. [Rok10]

The local nature of vertex smoothing leads to the need for optimising a cavity over and over again. After having smoothed a vertex, smoothing an incident vertex in the scope of its cavity may change the quality of the first cavity. Because of this property, the algorithm has to be applied a number of times in order to bring things to an equilibrium. The need for multiple iterations dictates that the local optimisation kernel should be computationally inexpensive. The process is summarised in Algorithm 2. Although the number of iterations to reach an equilibrium is given as a user-defined constant, it is not known a priori how many iterations will be needed and, in fact, it is not even guaranteed that the global optimum can be reached [FJP98]. Convergence criteria could also be used to terminate the algorithm, e.g. if the number of vertices smoothed during the last iteration is zero, then the loop terminates. Following subsections describe the algorithms used in this research to solve the local optimisation problem.

Algorithm 2 The vertex smoothing algorithm running MAX_ITER times.

```
Set S = {vertices that should be relocated};
int iteration = 0;
while(iteration < MAX_ITER)
{
    for(Vertex v : S)
        v.relocate();

    iteration++;
}
```

2.1.6 Laplacian Smoothing

Laplacian Smoothing is one of the most common smoothing algorithms, which just places the vertex to the geometric centre of all vertices surrounding the cavity. Denoting a vertex's coordinates as (x_i, y_i),

the new position for a central vertex in a cavity, defined by N surrounding vertices, is the arithmetic mean of all surrounding vertices:

$$x_{central} = \frac{1}{N} \sum x_i, \forall (surrounding\ vertex)_i$$

$$y_{central} = \frac{1}{N} \sum y_i, \forall (surrounding\ vertex)_i$$

Although computationally cheap, this method does not yield optimal results (it can actually worsen mesh quality) and, what is worse, can even end up returning an invalid cavity, in which some elements have been inverted and have negative area [Fre97]. Invalidation comes as a result of violating the *Interior Convex Hull* restriction. This convex hull of a cavity is an area within which relocation has to be restricted. An example of a cavity and its convex hull is depicted in Figure 2.6.

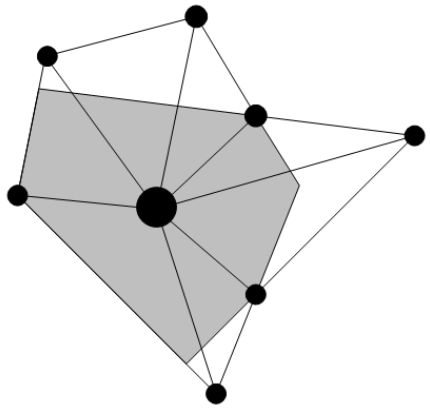

Figure 2.6: Example of an cavity and its interior convex hull. If the vertex under consideration is relocated outside the grey zone, some elements will have negative area and the mesh will be invalid.

The problems of Laplacian Smoothing can be avoided using *Optimisation-based smoothing*, a family of algorithms which try to optimise an objective functional, retaining validity of meshes and guaranteeing that the resulting cavities will be at least of the same quality as the initial ones. All this comes at the cost of heavier computational load, which is not necessarily a bad thing, as long as we are considering parallel implementations where, as it is known, heavy kernels favour scalability (of course, this does not mean that they also favour performance!). Therefore, optimisation-based smoothing is a perfect candidate for floating-point capable and massively parallel architectures like CUDA. The optimisation-based algorithms used in this research are described in the following sub-sections.

2.1.7 Algorithm by Freitag et al.

Vertex smoothing is a process which aims at maximising the quality of the worst element in a cavity. Recall that the quality of an element is measured using some objective functional, like the one proposed by Vasilevskii and Lipnikov [VL99]. The implied functional for the whole cavity can be defined as the even infinity norm of all quality functionals associated with the elements of this cavity. This description defines an optimisation problem for each individual cavity in the mesh. The following is a description of the optimisation problem and the solution proposed by [FJP95], as it is described in our previous work [Rok10].

"If we symbolise the cavity as C_i, its interior convex hull as H_i and the set of all elements inside C_i

as A_i then the function we want to maximise is:

$$\phi\left(\mathbf{p}\right) = min\left(Q_{M_{(\Delta)}}(\mathbf{p})\right), \forall \Delta \in A_i$$

A projection on the x-field of an example $\phi(\mathbf{p})$ is shown in Figure 2.7. As can be seen in that figure, each $Q_{M_{(\Delta)}}(\mathbf{p})$ is a smooth and differentiable function. Moving along a line of $Q_{M_{(\Delta)}}(\mathbf{p})$, there are more than one elements Δ that tend to obtain the minimum value. We say that these elements form the *active set A*. The active set, however, changes at the points denoted by black circles in the figure and at these points $Q_{M_{(\Delta)}}(\mathbf{p})$ is non-differentiable. In order to find the position $\mathbf{p}$ of the central vertex for which the worst element quality is maximised, we have to solve the non-smooth optimisation problem

$$max(\phi(\mathbf{p})), \mathbf{p} \in H_i$$

This problem has a solution at some point $\mathbf{p_s}$ if all directional derivatives of $\phi(\mathbf{p})$ at $\mathbf{p_s}$ are non-negative. This solution is unique because all functions $Q_{M_{(\Delta)}}(\mathbf{p})$ are monotonic while we move towards a fixed search direction. The non-smooth problem can be solved using a technique similar to the *Gradient Descent* method. This method, also called *Steepest Descent*, is used when we deal with twice-differentiable functions. The modified algorithm for non-smooth functions is the one described in Algorithm 3.

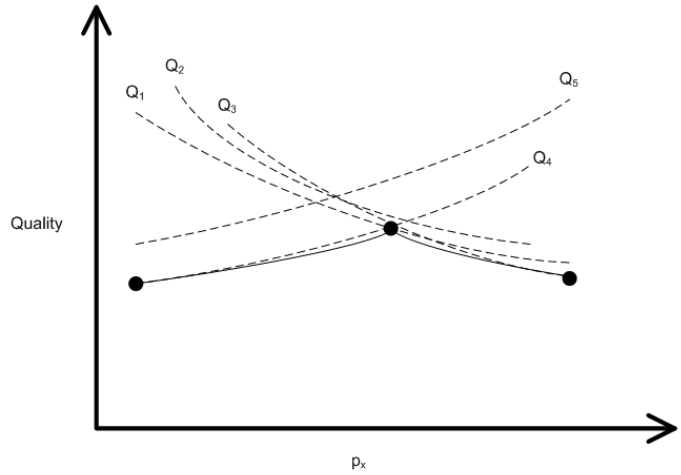

Figure 2.7: Projection of an example $\phi(\mathbf{p})$ on the x-field.

Let the original position of the central vertex be $\mathbf{p_{init}}$. At first, the algorithm calculates the interior convex hull of the cavity under consideration and chooses a starting point $\mathbf{p_0}$. If $\mathbf{p_{init}}$ does not coincide with $\mathbf{p_s}$, in which case we already have the optimal location for the central vertex and the algorithm exits, then a first guess for $\mathbf{p_0}$ is the geometric centre $\mathbf{p_c}$ of this hull. If this guess coincides with $\mathbf{p_s}$ then the algorithm stops. Recall that the criterion to determine whether $\mathbf{p_s}$ has been found is that all directional derivatives of $\phi(\mathbf{p})$ at this guess-point are non-negative. If $\mathbf{p_c}$ does not coincide with $\mathbf{p_s}$ then the algorithm decides which point $(\mathbf{p_{init}}, \mathbf{p_c})$ corresponds to a larger value for $\phi(\mathbf{p})$ and sets it as the starting point $\mathbf{p_0}$.

Having found a starting point, the algorithm iteratively tries to find a solution. At every estimation-point $\mathbf{p_i}$ the algorithm calculates all directional gradients. The search direction, i.e. the "overall" steepest direction, is calculated by taking the gradients of all $Q_{M_{(\Delta)}} \in A_i$ and finding all possible convex linear combinations of them (the latter implies solving a respective quadratic programming problem). After that, the algorithm has to decide how to move along the search direction, i.e. solve the "line search sub-problem". This task is accomplished by predicting the points at which the active set A_i will change. This prediction can be made by taking the first-order Taylor series approximation of

$Q_{M_{(\Delta)}}(\mathbf{p}), \forall Q_{M_{(\Delta)}} \in A_i$ and calculating the intersection of each such approximation with the projection of $\phi(\mathbf{p})$ in the search direction. The distance between $\mathbf{p_i}$ and the intersection point closest to it is the initial step length. If moving $\mathbf{p_i}$ along the search direction by this length improves $\phi(\mathbf{p})$ by {the estimated improvement ± some percentage} ([FJP95] propose ±10%) then this length is accepted and $\{\mathbf{p_i} + \textit{step\_length} \times \mathbf{search\_direction}\}$ becomes the new estimation-point $\mathbf{p_i}$. Otherwise, the step length is halved again and again until either a suitable length is found or the step length becomes smaller than a user-defined threshold. Additionally, a step is accepted on the condition that the next step improves $\phi(\mathbf{p})$ by a smaller amount.

The process described in the paragraph above is repeated until one of the following conditions are met:

1. $\mathbf{p_i}$ coincides with $\mathbf{p_s}$
2. the step length becomes smaller than the user-defined threshold
3. the improvement of $\phi(\mathbf{p})$ between two successive steps falls below a user-defined threshold
4. the number of iterations exceeds a pre-defined value"

2.1.8 Algorithm by Pain et al.

When solving the optimisation problem, we can also use non-differential methods, like the one described by [PUdOG01]. Recall that the optimal cavity is the one in which all triangles are equilateral with edges of unit length with respect to some error metric. In essence, the algorithm described here is *Laplacian Smoothing* in metric space.

Given a cavity C_i, it consists of a central vertex V_i and all its adjacent vertices V_j. Let L_i be the set of all edges the one end of which is the central vertex and the other end is one of the adjacent vertices. The aim is to equate the lengths of all edges $\in L_i$. The length of an edge l in metric space is defined as $r_l = \left(u_l^T M_l u_l\right)$. In order to have (as much as possible) equilateral triangles, we want to minimise the following functional:

$$E_i = \frac{1}{2}\sum_{l \in L_i} r_l^2 = \frac{1}{2}\sum u_l^T M_l u_l$$

If this functional is differentiated with respect to the position p^i of the central vertex and expressing the edge length in the standard Eucleidian space as $u_l = p^i - y_l^i$, where y_l^i is the position of an adjacent vertex V_j, then

$$\frac{\partial E_i}{\partial p^i} = \sum_{l \in L_i} M_l u_l$$

At the minimum of the above functional:

$$\sum_{l \in L_i} M_l p^i = \sum_{l \in L_i} M_l y_l^i - q^i = 0$$

where $q^i = \sum_{l \in L_i} M_l y_l^i$. If we define $A^i = \sum_{l \in L_i} M_l$, then we can calculate the position p^i using the equation $A^i p^i = q^i$. In order to ensure diagonal dominance and insensitivity to round-off error, a new diagonal matrix D^i is introduced so that the last equation can be written as

$$(D^i + A^i)(p^i - \hat{p}^i) = q^i - A^i \hat{p}^i$$

Algorithm 3 The optimisation-based smoothing algorithm by Freitag et al. [FJP95].

```
Cavity c = {cavity under consideration};
Function phi = {function phi(p)};

Point pInit = c.getCentralPoint();
Function gradPhi = phi.getGradientFunction();

if(gradPhi.isNonNegativeInAllDirectionsAt(pInit))
    return;

Point p = c.getConvexHull().getGeometricalCentre();
if(gradPhi.isNonNegativeInAllDirectionsAt(p))
{
    c.setCentralPoint(p);
    return;
}

if(c.testMinimumAngleWithCentre(p) <
   c.testMinimumAngleWithCentre(pInit))
    p = pInit;

int iteration = 0;
double step = INF;
while(!gradPhi.isNonNegativeInAllDirectionsAt(p) &&
      iteration < MAX_ITERATIONS && step > MIN_STEP_LENGTH &&
      c.getLastImprovement() > MIN_IMPROVEMENT)
{
    Direction searchDirection = c.findSteepestDirectionAt(p);
    step = p - c.getClosestIntrPointInDirection(searchDirection);

    while((c.testMinAngleImprAt(step, searchDirection) <
           0.9 * c.getEstimatedImpr() ||
           c.testMinAngleImprAt(step, searchDirection) <
           c.testMinAngleImprAt(step / 2, searchDirection))
          && step > MIN_STEP_LENGTH)
    {
        step /= 2;
    }
    if(step > MIN_STEP_LENGTH)
        p += step * searchDirection;

    iteration++;
}

c.setCentralPoint(p);
```

where $\hat{p}^i$ is the initial position of the central vertex and the diagonal matrix is defined as

$$D^i_{jk} = \begin{cases} maxA^i_{jj}, (1+\sigma)\sum_{m=1,m\neq j} \mid A^i_{jm} \mid, & if\ j = k \\ 0, & if\ j \neq k \end{cases}$$

The value of σ is taken to be $\sigma = 0.01$ in this project. Finally, it is important to use relaxation of p^i for consistency reasons, using $x^i = wp^i + (1-w)p^i, w \in (0,1]$. In this project, $w = 0.5$ and the new position of the central vertex can be found by solving the the equation

$$(D^i + A^i)(x^i - \hat{p}^i) = w(q^i - A^i\hat{p}^i).$$

In the case of boundary vertices, i.e. vertices which are allowed to move only along a line (the mesh boundary), a modification of the above algorithm has to be used. The restriction that the vertex can only move along a line means that the new position x^i can be calculated using the equation

$$x^i = a^i_C u^i_l + \hat{p}^i,$$

where u^i_l is the unit vector tangent to the boundary line and a^i_C is the displacement along this line measured from the initial position $\hat{p}^i$ of the vertex. a^i_c can be calculated from the equation

$$(D^i + \hat{M}^i)a^i_c = wg^i,$$

where

$$\hat{M}^i = u^{iT}_l \sum_{l\in L_i} M_l u^i_l$$

and

$$g^i = \sum_{l\in L_i} u^{iT}_l M_l (x^i - \hat{p}^i).$$

2.1.9 Rescaling the metric tensor

The main characteristic of vertex smoothing is that it does not change mesh topology, i.e. the number of mesh elements remains constant and the mesh is not coarsened or refined. In this case, because we cannot refine or coarsen the mesh, it is important to rescale the metric tensor field so that the expected number of elements, as defined by the metric tensor, is equal to the current number of elements. The expected number of elements can be defined as:

$$E_{new} = \frac{\sum_{e=1}^{E_{old}} A^\Omega_e}{\theta}$$

where E_{old} is the current number of mesh elements, $A^\Omega_e = A_e\sqrt{\det(\mathbf{M}_e)}$ is the area of a triangle in metric space (A_e is the area of this element in the standard Eucleidian space) and $\theta = \frac{\sqrt{3}}{4}$ is the area of an ideal element in metric space. If $E_{new} \neq E_{old}$, then the metric $\mathbf{M}$ has to be rescaled, so that we end up with a new metric $\mathbf{M}_{new} = \beta\mathbf{M}$. β is a scalar value which must satisfy the following condition:

$$E_{new} = \frac{\sum_{e=1}^{E_{old}} A_e\sqrt{\det(\beta\mathbf{M}_e)}}{\theta},$$

which gives that

$$\beta = \frac{\theta E_{new}}{\sum_{e=1}^{E_{old}} A_e\sqrt{\det(\mathbf{M}_e)}}.$$

2.2 Parallel Execution

This section gives a description of how vertex smoothing can be run in a parallel fashion, based upon the parallel framework proposed by [FJP98]. Parallel execution means that data are distributed over participating processing units. It is important to ensure *correct execution*, i.e. retain consistency of distributed data and get the same results out of the parallel algorithm as if we ran a serial one. In order to satisfy this requirement, we use the concepts of *elemental operations* and the *operation task graph*, a graph with respect to which the elemental operations must be synchronised.

Trying to ensure data consistency is one of the main reasons why parallel execution performance can be hindered. Being very strict on this consistence requirement would lead to a degeneration of the parallel algorithm into a sequential one. [FJP98] introduce the concept of *elemental operations* and propose that data consistency is maintained only *between* successive executions of these operations and *not during* their execution. This requirement leads to the formulation of the elemental operation steps:

(a) parallel execution of a set of some mesh improving techniques in each participating processing unit and

(b) global reduction between these units to update data modified by a.

Retaining data consistency essentially means three things. First of all, there cannot exist two processing units sharing ownership of the same data. Secondly, every mesh vertex has to know exactly what its neighbouring vertices are, i.e. which these vertices are (e.g. their IDs) and their position on the mesh. Neighbouring relationship has to be reciprocal, i.e. if processor 1 knows that its vertex v_1 has as neighbour processor 2's vertex v_2, then processor 2 must also know that v_2 has v_1 as neighbour. Finally, every mesh element unit has to know its adjacent elements as well. Once again, this relationship has to be reciprocal. A more formal formulation of these requirements is given in [Rok10] based on the proposal by [FJP98]:

"The type of data structures depends on the problem under investigation, the algorithm used and the specific implementation of this algorithm. The research group that proposed this parallel framework ([FJP98]), however, believe that the following properties must be fulfilled for every type of distributed data structures used:

- Every piece of mesh data (vertices, edges and elements) is owned by a unique processing unit – no two processors can share ownership of the same data.
- Vertex data have to retain their consistency. After any elemental operation, every vertex v has to know which its neighbours are, i.e. $adj(v)$. This knowledge has to be consistent, i.e. a vertex u is neighbour of v if and only if $v \in adj(u)$ in the processing unit owning vertex u. In other words, G_V has to be consistent across all processors. Knowing a neighbour means knowing which vertex it is (for example its global index number) and its position on the mesh.
- Element neighbour data have to retain their consistency. After any elemental operation, every element t has to know which are its neighbours, i.e. $adv(t)$. In other words, G_T has to be consistent across all processors. This knowledge can be used to perform some important operations, for example calculating a quality metric of two neighbouring elements in order to decide whether an edge-flip will improve local quality."

2.2.1 Operation Task Graph

The operation task graph G is a graph the vertices of which represent the elemental operations that have to be accomplished and the edges represent dependencies between operations. If the input of operation op_2 depends on operation op_1 then vertices v_1 and v_2 are connected by an edge. Edges are undirected, i.e. if $(v_1, v_2) \in G$, then also $(v_2, v_1) \in G$. The task graph is an essential structure as it allows us to extract *independent sets* of operations that can be executed in parallel. After executing an independent set of operations, we have to update neighbouring data (vertices, edges and elements) of the operations' results. At this point it is guaranteed that the distributed data structure will be consistent, since all operations executed were independent from each other. The general algorithm can be seen in Algorithm 4.

Algorithm 4 General parallel algorithm for the mesh refinement process.

```
TaskGraph G = Problem.createTaskGraph();
Set S = new Set(G.getTasksToBeAccomplished());
while(!S.empty())
{
    Set R = new Set();
    while(!S.empty())
    {
        Set I = new Set(S.getIndependentSet());
        I.executeElementalOperationsOnAllElements();
        I.updateElements(I.getAdjacentElements());
        R.add(I.getAdjacentElements().spawnElementalOperations());
    }
    S.setEqualTo(R);
}
```

This algorithm consists of two loops. The outer loop is call the *propagation* loop, because it spawns new elemental operations to adjacent entities (e.g. cavities, elements etc.). In the case of vertex smoothing, for example, once a cavity C defined by the central vertex V_C has been optimised, the cavities defined by all vertices adjacent to V_C have to be re-optimised, since a change in cavity C's geometry may have affected their quality. As for the inner loop, the number of iterations performed depends on the task graph and, more importantly, the way independent sets are extracted from it. The nature of vertex smoothing implies that the elemental operations can run asynchronously and mostly require only one-to-one communication between processing units (for other optimisation algorithms a few global reductions would also be required). This property is very important in the scope of efficiency and scalability of a parallel application.

2.2.2 Vertex Smoothing Elemental Operation

Executing the algorithm in parallel, we cannot smooth arbitrarily any vertices simultaneously. As can be seen in Figure 2.8, when we smooth a vertex, all adjacent vertices have to remain at their old positions. This means that the independent set of vertices to be smoothed contains vertices that are not adjacent. After smoothing, all neighbouring vertices must have their adjacency lists updated with the new position of the smoothed vertex. Algorithm 5 describes this elemental operation. The task graph is the vertex graph G_V which is essentially the representation of the mesh itself, as can be seen in Figure 2.9.

Next up, having presented the concept of *Anisotropic Mesh Adaptivity*, the main algorithms involved in

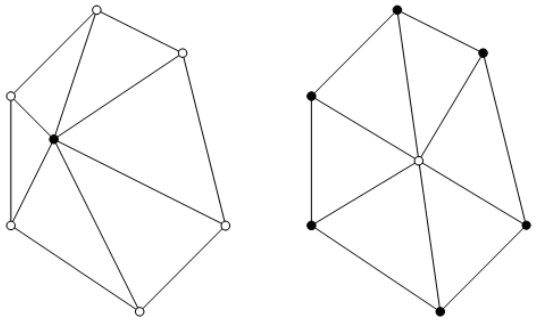

Figure 2.8: Example of how vertex smoothing should be performed. The left figure shows that in order to smooth the vertex denoted by a black circle we need to know the positions of all adjacent vertices (white circles). In the right, figure we see that all adjacent vertices must be updated with the new position of the smoothed one.

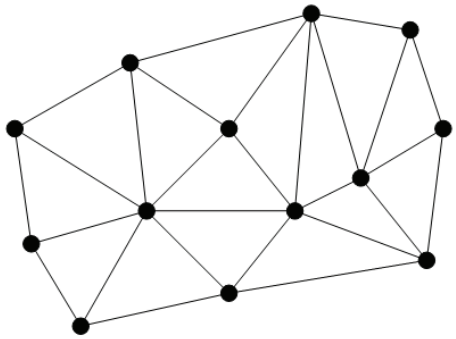

Figure 2.9: Example of a vertex graph G_V. If we ignore the exact shape of mesh elements, the same drawing could depict the mesh itself.

it and the way it can be run in a parallel fashion, we proceed to present nVIDIA's CUDA architecture, which is the platform of choice for this project, on which (platform) we attempted to implement all the above ideas.

Algorithm 5 Algorithm describing vertex smoothing elemental operation.

```
Vertex v = vertex under consideration;
v.smooth();
if(v.positionChanged())
{
    v.getAdjacentVertices().updateWithNewPosition(v.getPosition());
    R.add(v.getAdjacentVertices());
}
```

Chapter 3

nVIDIA's CUDA Architecture

nVIDIA's *Compute Unified Device Architecture* constitutes one of the most promising trends in modern processor industry. Over the past two decades, the evolution of 3D games pushed semiconductors industry to fabricate more and more powerful graphics processing chips. Gaming requirements have led to the advancement of GPUs to the point where GPU performance in floating–point arithmetic exceeded that of traditional CPUs. Up to recent years, all this tremendous processing power was intended to be used only for graphics and video processing purposes. The concept of *General Purpose GPU Computing* is a modern trend which aims at unleashing GPU processing capabilities and making them available to a wider range of compute intensive applications. A comparison between CPU and GPU floating-point capabilities over the past decade can be seen in Figure 3.1.

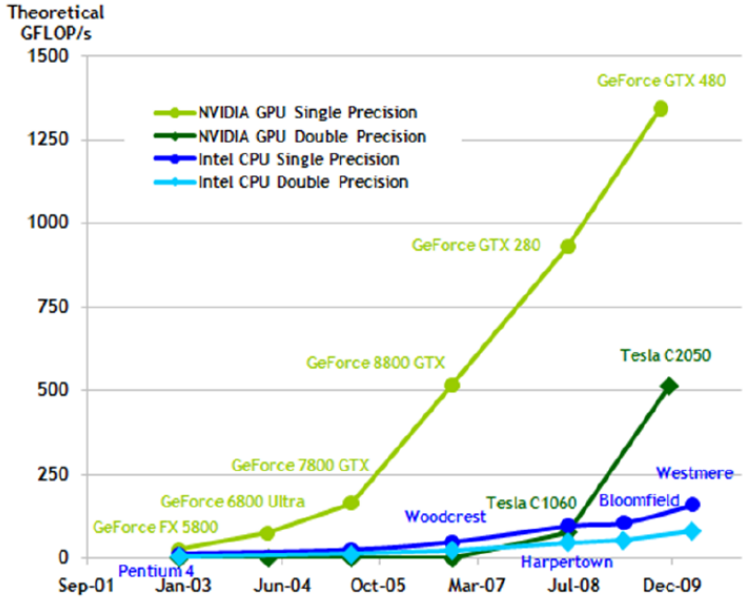

Figure 3.1: Comparison between the floating-point capabilities of conventional CPUs and GPUs over the years. (Figure from [nC10a])

Memory latency and limited bandwidth are generally considered to be two main bottlenecks in high-performance computing. High floating-point capabilities should be accompanied by high memory bandwidths and modern GPUs seem to have address this problem pretty well in comparison to traditional CPUs. Figure 3.2 shows the evolution of memory bandwidth over the past years.

CUDA is nVIDIA's attempt to enter the GPGPU market, which targets not only home and enterprise

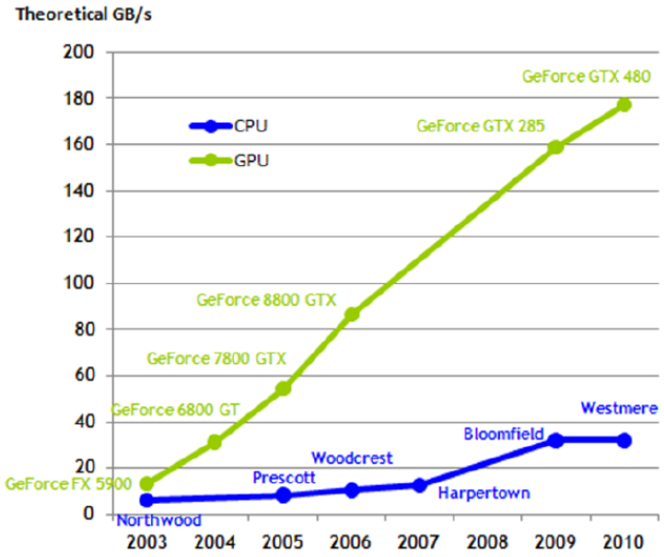

Figure 3.2: Comparison of memory bandwidth between conventional CPUs and GPUs over the years. (Figure from [nC10a])

users, but also the supercomputing field. The advantages of CUDA over conventional architectures lie on the way GPUs are designed to manipulate data. Contrary to a general purpose CPU, a graphics processor is designed to "apply the same shading function to many pixels simultaneously" or, in the case of GPGPU computing, "apply the same function to many data simultaneously" [cfd08]. Massive data processing means that GPUs can execute thousands of threads at the same time and are equipped with high bandwidth interconnection between processing cores and memory. This renders them perfect candidates for compute–intensive applications, but not for control–intensive tasks.

As can be seen in Figure 3.3, the main difference between a GPGPU and a traditional CPU is their floating-point capability. A GPGPU "is specialized for compute-intensive, highly parallel computation exactly what graphics rendering is about and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control" [nC10a].

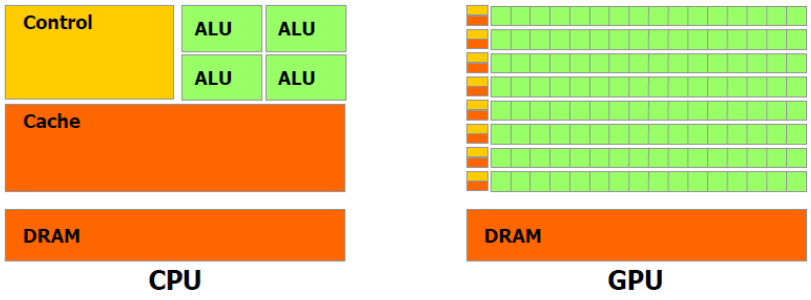

Figure 3.3: Traditional processing units devote transistors both to execution units and cache memories. CUDA, on the other hand, devotes more area to data processing. (Figure from [nC10a])

3.1 Architectural Overview

The philosophy behind CUDA programming is indissolubly bound with its massive data processing capability. The program is written to be executed by one thread and is finally instantiated multiple

times so that it is executed by many threads running in parallel. This model implies that the program has to be scalable in order to gain the most out of CUDA's hundreds of cores and thousands of threads. Depending on which GPU model is used, the number of available cores, therefore available execution threads, can vary. A general overview of this architecture can be seen in Figure 3.4 [cud08b]. The GPU consists of a number of *multi–processors*, each one containing 8 smaller *stream–processors*. Considering that a typical graphics card can have up to 30 multi–processors and each multi–processor can execute up to some hundreds of threads, a single GPU device can run simultaneously up to some thousands of threads.

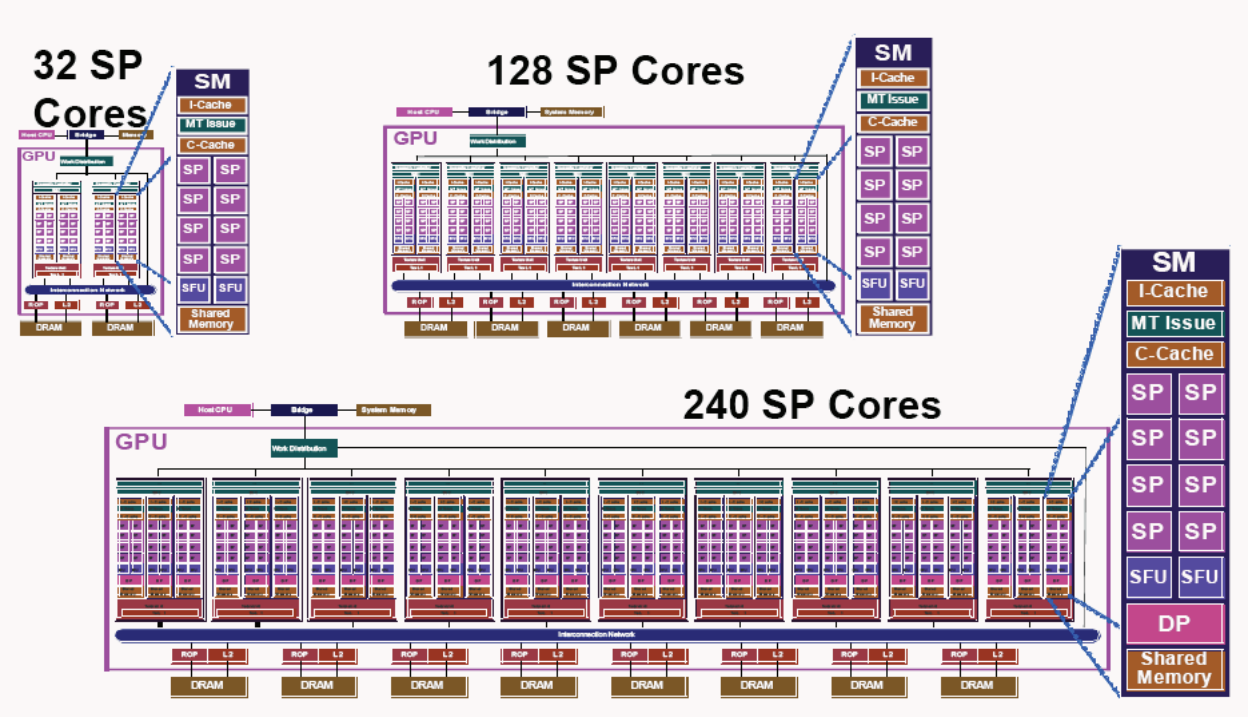

Figure 3.4: Overview of CUDA architecture. [cud08b]

3.1.1 Memory Model

A CUDA program consists of small pieces of code called *kernels*. Each kernel is executed simultaneously on many cores by multiple threads. All threads run exactly the same code, but may manipulate different data. The GPU communicates with the CPU through the PCI–Express bus. These two processing units have different address spaces, but data can be transferred between them. Explicit data transfer is mandatory and the CUDA API provides all necessary functions. Attempt to dereference a CPU address on the GPU or vice versa will likely result in a crash. Each thread has its own local storage (mainly in form of GPU core registers). Additionally, GPUs are equipped with 16KB per multiprocessor on–chip memory which is two orders of magnitude in terms of latency and one order of magnitude in terms of bandwidth faster than system memory. For devices of *compute capability* over 2.0, like the one used in this project (Fermi architecture), the on–chip memory is 64KB large and can be used both as software-managed shared memory and as a hardware-managed L1 cache. This memory can be shared between all threads of the same block but is inaccessible to any other thread. Figure 3.5 summarises memory hierarchy on a CUDA device and Table 3.1 describes properties of various memory levels [nC10a].

The graphics-processing roots of CUDA have left an important heritage which a GPGPU program can take advantage of: *Texture Memory*. Texture memory is writeable only from the host-side, whereas

Type	Location	Access	Scope
Registers	On–chip	R/W	Thread
Local	Off–chip	R/W	Thread
Shared	On–chip	R/W	Thread–Block
Global	Off–chip	R/W	Device & Host CPU
Texture	Off–chip	R	Device & Host CPU
Constant	Off–chip	R	Device & Host CPU

Table 3.1: Memory hierarchy on a CUDA device

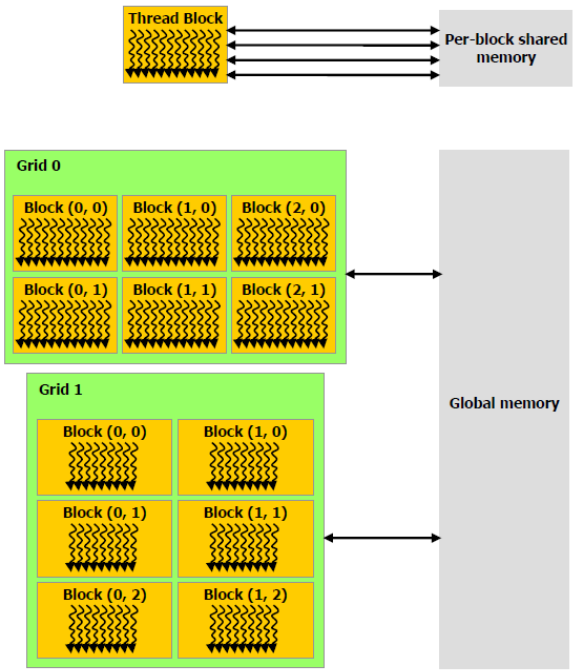

Figure 3.5: Memory hierarchy in a CUDA device. (figure from [nC10a])

a CUDA kernel can use in read-inly mode. Reading data from texture memory can have a lot of performance benefits, compared to global memory accesses. Texture memory is cached in texture cache, therefore reading texture data will involve a memory access only on a cache miss. Due to the nature of textures, the texture cache is optimised for 2D spatial locality. The main advantages of texture memory can be summarised in the following points:

- If memory accesses do not follow the patterns required to get good performance (as is the case with global memory), higher bandwidth can be achieved provided there is some locality on texture fetches.
- Addressing calculations are executed automatically by dedicated hardware outside processing elements, so that CUDA cores are not occupied by this task and the programmer does not have to care about addressing.
- Packed data can be broadcast to separate variables in a single operation.
- Integer data (8-bit or 16-bit) can be automatically converted to floating-point values in the range of [0.0, 1.0] or [-1.0, 1.0] [nC10a].

Textures are discretised data from a (theoretically) continuous domain. In graphics processing, there may be needed a texture value from a coordinate which falls between discretisation points. In this case, there has to be performed some kind of texture data *filtering*. The nature of graphics textures resembles the metric tensor field used in anisotropic mesh adaptivity problems. As was described earlier, the metric tensor is discretised vertex-wise. However, when moving vertices during the optimisation process, the new vertex location will most possibly not coincide with a discretisation point. If this is the case, the value of the metric tensor field can be found by interpolating the values from the four nearest discretisation points. This is completely analogous to the most common type of texture filtering, *Linear Filtering*. An example of linear filtering is shown in Figure 3.6.

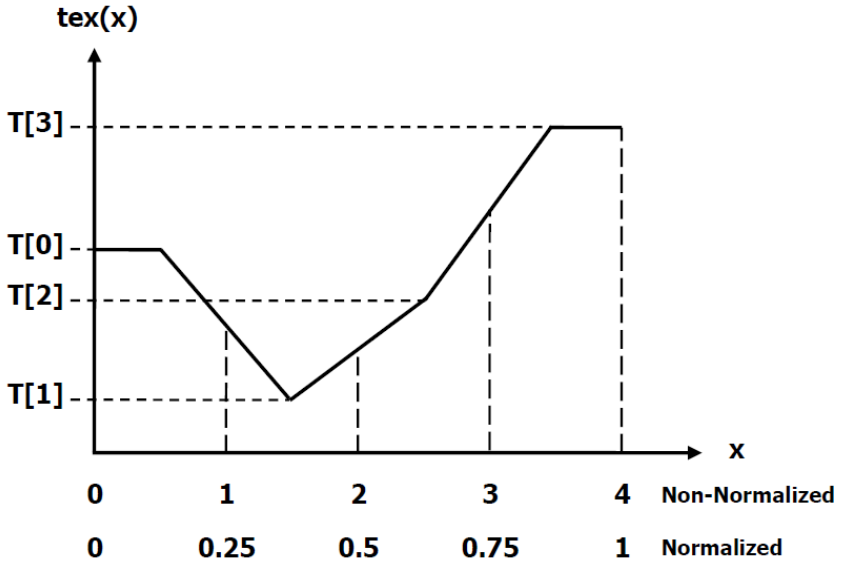

Figure 3.6: Example of 1D linear texture filtering. (figure from [nC10a])

In two dimensions, the result $tex(x, y)$ of linear filtering is

$$tex(x,y) = (1-\alpha)(1-\beta)T[i,j] + \alpha(1-\beta)T[i+1,j] + (1-\alpha)\beta T[i,j+1] + \alpha\beta T[i+1,j+1]$$

where α is the horizontal distance of point (x, y) from the nearest texture sample (discretisation point) $T[i, j]$ and β is the vertical distance. The key point is that, when using texture memory, this calculation is automatically performed by dedicated texture hardware outside multiprocessors. When using a conventional CPU or a GPU reading texture from global memory, the above calculation has to be performed by CPU / CUDA cores as part of the execution flow, which has to be programmed manually and occupies execution resources.

From the analysis above, we reach the conclusion that there are two issues of benefit when we make use of texture memory for the metric tensor field. Performance can benefit from texture caching as well as hardware implementation of interpolation. In the scope of this project, we did not evaluate the individual contribution of these two issues. This evaluation would be very interesting, especially when 3D adaptivity problems are considered, and is left as future work. In a 3D problem we have to interpolate the values of the 8 nearest neighbours:

$$\begin{aligned} tex(x,y,z) = {} & (1-\alpha)(1-\beta)(1-\gamma)T[i,j,k] + \alpha(1-\beta)(1-\gamma)T[i+1,j,k] + \\ & + (1-\alpha)\beta(1-\gamma)T[i,j+1,k] + \alpha\beta(1-\gamma)T[i+1,j+1,k] + \\ & + (1-\alpha)(1-\beta)\gamma T[i,j,k+1] + \alpha(1-\beta)\gamma T[i+1,j,k+1] + \\ & + (1-\alpha)\beta\gamma T[i,j+1,k+1] + \alpha\beta\gamma T[i+1,j+1,k+1] \end{aligned} \tag{3.1}$$

and doing so would require (compared to 2D problems) double the data volume to be fetched from global memory and more than double the floating-point operations if we do not use hardware implementation of interpolation.

3.1.2 Programming Model

A great advantage of programming on CUDA is that it is very similar to conventional C/C++ programming and it is fairly easy to port existing codebase onto the new platform. Even if writing entirely from the beginning, parallel programming on CUDA is much more effortless than on other platforms, e.g. developing applications for the Cell Broadband Engine require writing code specifically for Cell's execution cores. A CUDA kernel is executed by many threads. These threads are grouped into so called *thread blocks* and each thread within a block is identified by thread and block ID variables (akin to processor rank number in MPI). Using these identifiers it is possible to specialise the execution path of each thread. Blocks are executed one per multi–processor. The special thing about a block is that all threads within it can communicate via shared memory and synchronise with each other. All blocks form what is called a *grid*. Grids are executed one per GPU device. This description is illustrated in Figure 3.7 [nC10a]. IDs of blocks are 2–coordinate ones and each block's ID is unique within a grid. Following the same convention, IDs of threads are 3–coordinate ones and each thread's ID is unique within a block. Grid dimensions are instantiated during application launch [cud08a].

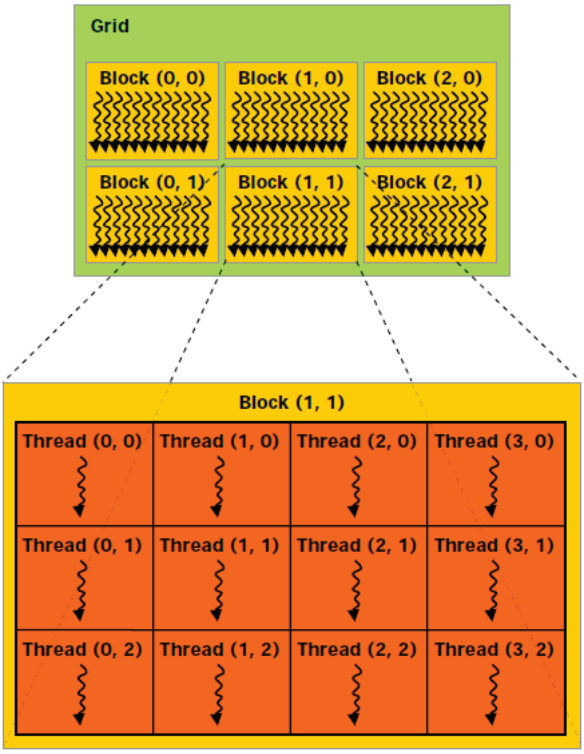

Figure 3.7: Threads and blocks in a CUDA device. [nC10a]

Specialising the execution path of threads, however, can degrade performance. If two threads within the same *warp* (a warp is a group of 32 threads that are created/managed/scheduled/executed in parallel) need to diverge their execution paths, then these executions will be serialised, i.e. the first path will be executed while the thread following the second path is stalled and after that the thread following the first path will be stalled while the second threads is executed.

An important principle in CUDA programming is that blocks must be independent from each other.

They can be executed in any order, in series or in parallel, and coordinate their execution. However, having shared locks between blocks to achieve coordination can not only lead to deadlocks but also degrade application scalability, the latter being destructive considering the philosophy behind CUDA. Synchronisation between CPU and GPU can be implemented using the CUDA API. When CPU code calls a kernel to be executed on a GPU, control returns to CPU immediately and the kernel is executed asynchronously. On the other hand, data transfer calls are blocking ones. There are non–blocking versions of these calls and this allows us to overlap computation and communication, therefore increasing performance and favouring scalability.

Additionally, there are calls which synchronise all threads within a block, i.e. they put a barrier at that particular point in execution which must be reached by all threads in the block before execution can continue [cud08a]. Unfortunately, blocks cannot synchronise with each other. The only way to synchronise two or more blocks is to wait for the kernel to finish execution and then re-launch it. This action definitely involves some overhead which can degrade performance. However, the cost is not detrimental, because CUDA threads are lightweight an invoking a kernel is much cheaper than creating, for example, an OS thread.

3.1.3 Execution Model

As was mentioned earlier, a CUDA program consists of small pieces of code called kernels. Kernels are instantiated many times and each instance is executed by a dedicated thread. Many threads form a thread–block which is executed on a multi–processor. One multi–processor may have many concurrent thread–blocks, the number of which depends on the available hardware resources, i.e. on–clip shared memory and register file. A thread–block is organised in *warps*, each one consisting of 32 threads and being executed in a SIMD fashion. Many thread–blocks form a grid, which is executed on a CUDA device. This device is dedicated to its grid, i.e. there cannot be two or more grids under execution on a device at the same time [cud08c].

3.2 Code Optimisations

In this subsection we will try to describe some general guidelines that maximise execution performance on a CUDA device. These ideas are basically what was proposed at [cud08c], which in turn is a comprehensive summary of nVIDIA's guidelines.

3.2.1 Memory Optimisations

A CUDA device is equipped with a memory system similar to other high–performance platforms, like the Cell Broadband Engine. General rules for these platforms also apply here. Bandwidth is much higher for inter–device communication than between host CPU and device GPU. Therefore, the latter communication should be minimised. Additionally, every data transfer has a certain amount of overhead, independently of transfer size, so one large transfer is preferable to many small ones. Communication between host and device can be enhanced by taking advantage of CUDA's ability to allocate *page–locked* system memory. This technique maximises bandwidth for memory copy operations. Moreover, the application should be written in such a way so that computation and communication are overlapped. This is possible on CUDA thanks to its asynchronous memory copy operations. Hiding communication latency behind computation not only favours performance but also application scalability to more cores and devices.

Data alignment in memory is also an important factor in achieving good bandwidth utilisation and high performance. Depending on the device's *compute capability*, an access to global (device) memory can be serviced by as few as 1 or 2 transfers. Accessing misaligned data, however, is followed by many more memory transactions and performance can be seriously degraded. A way to address this problem is utilisation of fast, on–chip shared memory, which can be used as a data re–alignment buffer for data to be stored back to global memory (apart from its obvious uses as a means of thread synchronisation within a block and global memory caching).

Shared memory accesses, however, have their own rules. This memory is divided into banks. Successive 32-bit words are allocated to successive banks. Each bank can be accessed by one thread at a time. If two or more threads try to access the same bank simultaneously, requests will be serialised. If there are no bank conflicts, shared memory can service simultaneously as many requests as the number of its banks. Access speed is the same as for the register file. As is the case with shared memory, global memory is also divided into *partitions*. Successive 256-byte sets of data are assigned to successive partitions. Partition access should follow a pattern similar to that of shared memory. If many threads try to access (either read of write) the same partition performance will be degraded and the phenomenon is called *partition camping*.

3.2.2 Execution Configuration Optimisations

As is the case with every other multi–threaded platform, instructions within a thread are executed sequentially, so if a thread stalls for some reason (data dependencies, data transfers) hardware utilisation is reduced. This problem can be addressed by running many warps on a multi–processor. General guidelines are the following:

- The number of thread–blocks must be larger than the number of multi–processors so that every multi–processor has at least one block to run.
- Even better, the number of thread–blocks should be al least twice the number of available multi–processors so that even if a thread–block stalls (e.g. waiting on memory), the multi–processor will continue to be busy executing the other block.
- If we want to take into account future GPUs which will incorporate even more multi–processors, the number of thread–blocks should be even larger than the previous case.

Additionally, thread–block size should be multiple of warp size so that there are no under–occupied warps that waste resources.

3.3 C++ support

Although nVIDIA provides full support for C code, C++ is only partially supported by the CUDA compiler. C++ features that can be used by devices of compute capability 2.0 (Fermi architecture) are the following:

- Polymorphism
- Default parameters
- Operator overloading

- Namespaces
- Function templates
- Classes

More specifically on class support, any class function can be compiled to CUDA object code as long as this function is not virtual (a restriction that will be removed in future architectures). Moreover, all functions are inlined by the compiler because CUDA hardware is not capable of calling functions. Unfortunately, the linker cannot link two functions that belong to different object files and this is the reason why all classes must reside in the same source file.

Having presented both the topic of *Anisotropic Mesh Adaptivity* and nVIDIA's CUDA architecture, which comprise the essential background the reader should be familiar with, we go on to the next chapter where we describe the design choices we had to make and the actual implementation of *CUDAMesh64*.

Chapter 4

Design and Implementation

This chapter contains a thorough description of the application which was developed as a way for us to evaluate the performance of anisotropic mesh adaptation. There is a description of the object-oriented model the code-base is built upon, the data structures that are used to represent all useful information about the adaptivity problem and the optimisations that were applied in our effort to improve CUDA performance. Design and implementation were guided to a great extent by [nC10a] and [nC10b]. At the end of the chapter, there is a list of problems that were encountered during the development and optimisation phases. Apart from benchmarking purposes, the application described in this chapter is also a stand-alone application, called *CUDAMesh64*, which can be used in the solution of real adaptivity problems.

4.1 Design choices

Before proceeding to the description of the actual implementation, it is necessary to list the design choices we had to make at the beginning of this implementation and the reasons why we took these specific decisions.

The first thing we had to specify was the target platform of this application. We chose to work on nVIDIA's *Fermi* architecture, i.e. devices with compute capability 2.0. Apart from being the state-of-the-art among CUDA architectures, *Fermi* offers better C++ support than previous generations, which is an important feature when it comes to code development and testing compared to pure C. Another advantage, and maybe the reason that renders *Fermi* mandatory choice, is its capabilities in double-precision floating-point arithmetic. Contrary to older compute capabilities, where double-precision data had to be broken down into pairs of single-precision values, *Fermi* overcomes this limitation and manipulates double-precision data just like it does with single-precision ones.

Double-precision arithmetic was chosen over single-precision because it is the standard choice in the world of scientific applications. The choice of higher precision is indicated by the need to make the application more robust to the order in which arithmetic operations take place (a quite common problem in numerical analysis) and reduce round-off errors. In addition, structuring our application on double-precision data gives us the chance to assess Fermi's double-precision capabilities and make more meaningful comparisons between CPU and GPU, since in today's CPUs even single-precision variables are represented internally as double-precision ones.

Mesh adaptivity can be carried out using different families of algorithms. The most common families are *h-adaptivity* and *r-adaptivity*. The first one includes algorithms which change mesh topology by adding vertices and edges, creating new elements by bisection or adaptive refinement, flipping

edges etc. [Rok10]. The family of *r-adaptivity* algorithms, on the other hand, includes techniques that leave mesh topology intact. *Vertex Smoothing* is an *r-adaptivity* algorithm. Not modifying mesh connectivity makes implementation much easier. Both the number of mesh vertices and their connectivity remain constant and so does the operation task graph, i.e. there is no need to re-colour the graph after every iteration over the mesh when running in parallel. Moreover, vertex smoothing is a computationally demanding technique and this property helps in showing off CUDA's capabilities in floating-point arithmetic and exhibiting large speedups over conventional platforms.

As was mentioned in the introductory chapter, there is a related project called *Fluidity*. We decided to build *CUDAMesh64* independently of *Fluidity*'s data structures, using our own ones. *CUDAMesh64* is not part of *Fluidity* but rather an independent component for a mesh bases simulator which can be coupled with any simulator. Having data structures customised for the specific problem of vertex smoothing would allow us to optimise the application in a better way and achieve higher performance.

As far as graph colouring is concerned, we chose to implement a single-threaded and greedy colouring algorithm, called *First Fit Colouring* [AOS06]. Although being a greedy algorithm, *First Fit* runs adequately fast and still colours the mesh with satisfactorily few colours. These properties, along with the simplicity of implementation, indicated that *First Fit* is just fine for the purpose of this project, so parallel algorithms or more sophisticated colouring techniques were not deemed necessary.

As will be described in later sections, one of the optimisations we can enable in our code is having dedicated texture hardware interpolate metric tensor values. This interpolation can be done using either *nearest-point sampling* or *linear filtering*. We chose to use the latter method, because nearest-point sampling can be very unsuitable, especially when the metric tensor field has a lot of discontinuities.

The whole application and all auxiliary frameworks used throughout this project are compiled and used in 64-bit mode, hence the name *CUDAMesh64* of the target application. The reason behind our choice of this mode is not that it is really needed at this time but because this framework was built with extensibility and scalability in mind. Problems become more complex and so do the data representing them. It is a matter of just a couple of years until hardware requirements of an adaptivity problem exceed the order of magnitude in which we worked in this project. There is already the *Tesla C2070* card which is based on *Fermi* architecture and hosts 6GB of device memory. Future architectures are expected to host amounts of memory well above the 32-bit limit of 4GB.

4.2 Meshes and the VTK framework

The first thing that was taken into consideration is the way meshes can be represented and stored. Due to its popularity and also because it is a very mature and reliable framework, the *Visualization Toolkit (VTK)* was employed [Kitb]. As its creators describe it, the VTK is an open-source, freely available software system for 3D computer graphics, image processing and visualization, which consists of a C++ class library and several interpreted interface layers.

Unstructured meshes are stored using this framework in VTK's XML unstructured grid files. This type of file is in essence an XML description of the mesh, i.e. the vertices comprising the mesh, their coordinates, the triplets of vertices forming mesh cells (triangles) and any field data. The XML parser provided by the framework reads in all important information from the VTK file and passes them to the core of the adaptivity application. After the process of adaptation is done, the core returns the new coordinates of all mesh vertices which are written to a new VTK file.

To visualise the results we use Kitware's *ParaView* [Kita]. ParaView is an open-source, multi-platform data analysis and visualization application which can build visualizations in order for data to be analysed using qualitative and quantitative techniques. The data exploration can be done interactively in

3D. ParaView was developed to analyse extremely large datasets using distributed memory computing resources; however, in this project it was only needed for data visualisation on a single computer.

4.3 The object-oriented model and data structures

4.3.1 MeshOptimizer class

CUDAMesh64 can be considered as a unit test for the adaptive framework described in this section. The application consists of one main execution source file, which parses the command line arguments (input VTK file, number of iterations, platform to be executed on (CPU or CUDA) and, in case of CUDA, the number of threads per block). The main file creates thereafter a *MeshOptimizer*, an object which parses the VTK file, passes all useful data to a *Mesh* constructor (which builds the mesh using the data structures of our choice), uses a graph colouring algorithm to build independent sets out of this mesh, invokes the optimisation process and, finally, writes the new vertex coordinates to an output VTK file.

In terms of benchmarking, this class is responsible for initiating timers, measuring time differences and printing timing results to the standard output, so that the user can see exactly how long the optimisation process took. It should be noted that we are only interested in the optimisation process and time measurements exclude parsing of VTK files, construction of the mesh, mesh colouring and result writeback.

4.3.2 Mesh class

The *Mesh* class consists of four main parts: an array of *Vertices*, an array of *Elements*, an array of *Cavities* and a structure which holds all information about the independent sets, which are extracted from the mesh so that the optimisation process can be correctly executed in parallel. The constructor of this class is responsible for converting the information provided by the VTK file to our inner representation of choice, i.e. it takes all information about grid points and converts them to *Vertex* information, all information of grid cells and converts them to *Elements* and, based on cell data, it creates a vertex adjacency list, which is used both for construction of *Cavities* and mesh colouring.

Vertices and elements are stored in the mesh using vertex- and element- IDs. An ID is an ascending number, beginning from 0, which corresponds in a one-to-one fashion to the order in which data are read from the VTK file. This makes storing the new vertex positions back to the grid easy.

Another responsibility of this class is to copy all this information to the address space of the GPU. The initial implementation of this class was based on STL *std::vector* containers which, according to the specification, store elements in contiguous memory addresses, making the process of copying a vector between host and device very simple and completing it in one transfer. This choice, however, proved to be unsuitable because CUDA does not support STL; therefore *std::vector*'s member functions could not work on the device. A formal and apparently safe solution would be using the *Thrust* library [HB10], a library developed by nVIDIA programmers. As its creators claim, "Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). Thrust provides a flexible high-level interface for GPU programming that greatly enhances developer productivity" [HB]. Although this library was taken very seriously into consideration, the fact that it is still an early effort and there is not much feedback about it discouraged us from using it. We preferred the solution of converting all *std::vector* containers into standard C arrays.

4.3.3 Vector2d structure

In the base of this application lies the *Vector2d* structure. A *Vector2d* is a data structure representing a 2D vector. It contains a pair of coordinates, stored as two double-precision floating-point values, i-coordinate and j-coordinate. For ease of programming, *Vector2d* provides overloaded methods for:

- Addition and subtraction of two *Vector2d*s
- Multiplication and division of a *Vector2d* by a scalar value
- Multiplication between two *Vector2d*s (inner product)
- Normalisation of a *Vector2d*
- Checking whether two *Vector2d*s are co-linear

All these methods can be compiled both to host and device code.

4.3.4 Vector2dPair structure

A *Vector2dPair* is a data structure which contains two *Vector2d*s, i-Vector and j-Vector. Essentially, *Vector2dPair* is a 2×2 matrix. As was the case of *Vector2d*, *Vector2dPair* provides overloaded methods for:

- Addition of two *Vector2dPair*s
- Multiplication of a *Vector2dPair* by a scalar value
- Multiplication between a *Vector2dPair* and a *Vector2d*, which results to a new *Vector2d*

which can be compiled to both architectures, host and device.

4.3.5 Vertex class

Every mesh vertex is represented in *CUDAMesh64* by a *Vertex* object. Every instance of this class contains just a *Vector2d*, which represents the coordinates of its vertex in the mesh. There are also methods for retrieving vertex coordinates and setting new ones. All these methods can be used both from host and device code.

4.3.6 Element class

An *Element* is a mesh triangle (this class it is called *Facet* in the codebase). Every *Element* contains the IDs of the three vertices comprising it. There are methods for retrieving and setting vertex IDs, as well as methods which return the important geometric characteristics of a triangle, i.e. its area and perimeter with respect to a metric tensor field. This class can be fully used from both host and device code. It should be noted that, as was the case with the *Mesh* class, vertex IDs were initially stored using STL vector containers, which were later dropped and replaced by C-style arrays.

Additionally, in the very early development phase it was pointers to the actual *Vertex* objects that were stored in the vertex list, not IDs. This was expected to speed up execution performance, because

a *Vertex* object could be accessed directly using this pointer, instead of using the vertex ID to access the corresponding index in the array of vertices inside the *Mesh*. However, there is no point in copying pointers between host and device address spaces, so the pointer-to-vertex approach was abandoned.

4.3.7 Cavity class

As was mentioned in 2, a cavity is defined by its central vertex, all adjacent vertices and all elements defined by these vertices. Therefore, a *Cavity* object is a data structure containing the vertex and element IDs of all its surrounding vertices and elements, respectively. Initially, a *Cavity* object would contain STL vectors of pointers to vertices and elements, but for the reasons mentioned earlier we followed the approach of C-style arrays of IDs. Another property of a cavity is whether it is defined by a boundary vertex, because boundary vertices are smoothed using different variations of the main smoothing algorithms. The *Cavity* class also implements methods for setting and retrieving the IDs of surrounding vertices and elements, which can be used both from host and device code.

4.3.8 Metric class

One of the most important structures in anisotropic mesh adaptation is the metric tensor field. CUDAMesh64 implements two versions of metric tensors, a *Continuous Linear Metric* and a *Discrete Linear Metric*. The difference between these two versions, as their names imply, is that the latter actually contains a discretised metric, i.e. a matrix containing metric samples from various points, whereas the former is implemented as a continuous function which returns the metric value at the requested point.

For the continuous metric test cases, we considered a linear variation across the domain: the expected element size in both x- and y- directions grows linearly from *minValue* to *maxValue* as we move along each direction. This means that the metric tensor (which is a *Vector2dPair* object) reaches its smallest value in

$$T[0.0, 0.0] = \begin{bmatrix} \frac{1}{maxXValue^2} & 0 \\ 0 & \frac{1}{maxYValue^2} \end{bmatrix}$$

and its largest value in

$$T[1.0, 1.0] = \begin{bmatrix} \frac{1}{minXValue^2} & 0 \\ 0 & \frac{1}{minYValue^2} \end{bmatrix}.$$

Using these metric tensor fields, the ideal mesh should appear having an increasing density of elements as we move from [0.0, 0.0] to [1.0, 1.0]. An example can be seen in Figure 4.1 and Figure 4.2. In this example, an initial Delauney mesh is being adapted with respect to a linear metric, the values of which lie in the range [0.05, 0.5] in both directions. Note in this example that in the lower left corner, where the metric value is low, i.e. the metric space is similar to the standard Eucleidian space, triangles are almost equilateral and begin being stretched as we move towards right and up.

For testing, *CUDAMesh64* also contains a second continuous metric field, the *Sinusoidal Metric*. Using this metric, the density of elements follows a sine curve as we move from left to right. The result is shown in Figure 4.3. New metrics can be easily added to *CUDAMesh64*, as long as they comply to the interface the rest of the application expects from a metric to expose.

The discrete version of metrics has been implemented as a 2D matrix of *Vertex2dPair*s. Normally, the metric tensor field is discretised vertex-wise. In *CUDAMesh64*, however, metric values are stored as

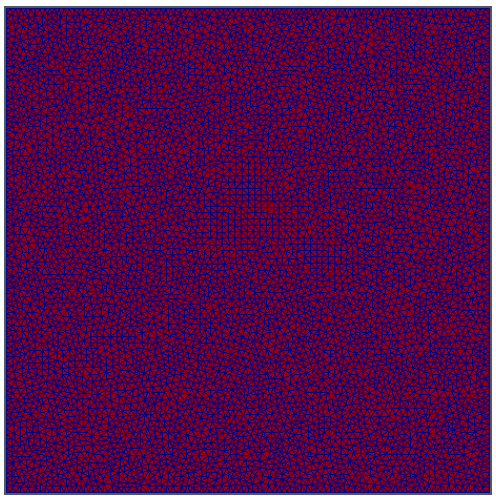

Figure 4.1: A Delauney mesh before anisotropic adaptation

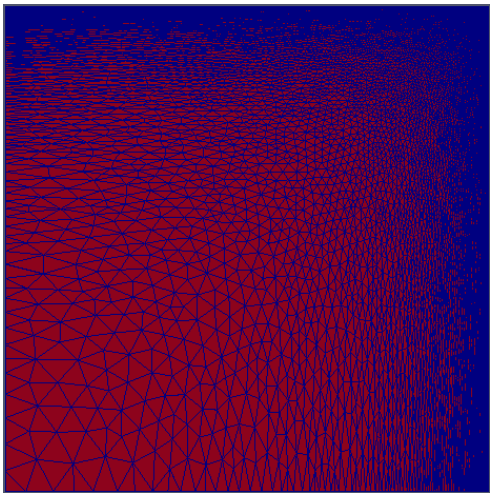

Figure 4.2: The mesh from Figure 4.1 after anisotropic adaptation using the linear metric tensor field, with $minValue = 0.05$ and $maxValue = 0.5$ in both directions.

if they were sampled on a uniform, structured mesh. In order to preserve accuracy, the metric tensor field is super-sampled, meaning that the dimension of this matrix is equal to the next power of 2 that is greater than the mesh size, e.g. a mesh that consists of 100×100 elements will be adapted using a 128×128 metric matrix. The choice of a structured metric representation was indicated by the optimisation we had in mind to manipulate metric data as textures.

4.3.9 ObjectiveFunctional class

The *ObjectiveFunctional* class implements the objective functional by Vasilevskii & Lipnikov, which was described in 2.1.2. This class implements only one method, which takes one argument indicating which cavity is to be optimised. The evaluation function can access the *Mesh* in order to retrieve information about cavities, elements and vertices and uses the methods provided by the *Element* class in order to evaluate the quality of the cavity under consideration with respect to a given metric tensor field. The result of this evaluation is the value of Vasilevskii's & Lipnikov's functional for the worst element in the cavity.

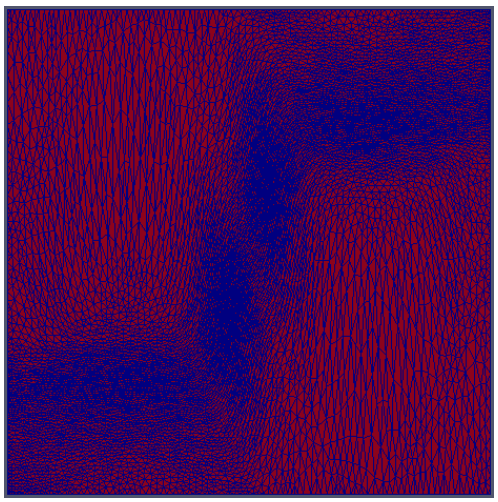

Figure 4.3: The mesh from Figure 4.1 after anisotropic adaptation using the sinusoidal metric tensor field, with $minValue = 0.05$ and $maxValue = 0.5$.

This class is designed to be used by the optimisation algorithm by Freitag et al. (2.1.7) which, as is described in the last section of this chapter, was not possible to be implemented. Using an *Automatic Differentiation (AD)* tool, the gradient of the evaluation method can be used by this algorithm to find the optimal position of a central vertex. Although an appropriate AD tool was not available at the time of *CUDAMesh64* development, all necessary work has been done so that, when such a tool becomes available, it can use this class directly to produce the gradient of the evaluation function, making the integration of AD into *CUDAMesh64* very easy.

4.3.10 OptimizationAlgorithm namespace

The *OptimizationAlgorithm namespace* contains functions which perform the actual mesh optimisation procedure. As far as anisotropic adaptivity is concerned, this namespace contains only the implementation of the optimisation algorithm by Pain et al. (2.1.8), leaving the algorithm by Freitag et al. (2.1.7) as future work, when an appropriate AD tool can be found. *Laplacian Smoothing* (2.1.6) has also been implemented, but it will not be analysed, since it is of limited practical interest in an anisotropic problem and may generate inverted elements.

The optimisation function receives three arguments: a pointer to the mesh to be optimised, a pointer to the metric tensor field used in the anisotropic adaptation process and a number of iterations to be performed. As was explained in 2.1.5, the number of iterations needed when performing vertex smoothing can be limited by some convergence criteria and a maximum number of iterations is provided by the user to guarantee termination. This function examines each cavity in turn. Depending on whether the cavity is defined by an inner or a boundary vertex, the optimisation function calls the appropriate variant of Pain's algorithm: functions *relocateInnerVertex(...)* and *relocateOuterVertex(...)*, respectively.

The same namespace also contains the corresponding version of the optimisation method for CUDA. This version takes a fourth argument of type *dim3*, indicating the number of CUDA threads per block. The same method is responsible for copying the metric tensor field from the host address space to the device address space.

4.4 Parallel implementation

This section describes the two basic tools which are essential for the correct parallel execution of anisotropic mesh adaptivity: the *IndependentSets* class and the *GraphColoring* namespace. After their presentation, it is explained how the independent sets are used to run the application in parallel using *OpenMP* or a *CUDA* device.

4.4.1 IndependentSets class

An independent set of a graph contains vertices which are not connected through and edge with each other in the graph. This property allows us to optimise cavities defined by vertices from the same independent set simultaneously, without disrupting correct execution, as was described in 2.2. It is obvious that the larger the independent sets, the more parallelism we can extract from *CUDAMesh64*. Large independent sets, each containing thousands of vertices, are necessary in order to exploit CUDA's massive parallelism capabilities.

The *IndependentSets* class is a structure which holds information about the independent sets of a mesh, as they have been extracted by a graph colouring algorithm. Essentially, an *IndependentSets* object contains an array of pointers to arrays of vertex IDs. Each such array contains the IDs of all vertices that belong to the same independent set. This class is accompanied by auxiliary structures which assist in copying the independent sets from host to device memory.

4.4.2 GraphColoring namespace

GraphColoring is a namespace which contains the implementation of the graph-colouring algorithm. In this project, only one colouring method was implemented, the *First Fit Colouring* (greedy) algorithm [AOS06]. This method considers vertices in ascending ID order and assigns to each vertex the smallest available colour not occupied by any of the adjacent vertices, adding a new colour if necessary. Colouring of one vertex is described in Algorithm 6. Algorithm 7 describes the process of colouring the entire mesh. In the latter algorithm, it can be seen that vertices are considered in ascending ID order and each one is coloured by probing the 0th colour as the first colour to be tested.

4.4.3 Parallel execution using OpenMP and CUDA

Executing *CUDAMesh64* in parallel on the host CPU can be done by using *OpenMP*. Parallelising the optimisation function can be done very easily by executing Algorithm 8. Instead of considering vertices one at a time, this function can be implemented as a for-loop, in every execution of which an independent set in considered. Processing of all vertices inside this set is parallelised by putting the appropriate OpenMP directive before the inner for-loop that traverses the set.

On the other hand, parallel execution on the device can be done as described in Algorithm 9. Once again, there is a for-loop which considers one independent set at a time. Having provided the number of threads per CUDA block as an input argument in *CUDAMesh64*, the algorithm launches as many blocks as needed so that every vertex in the independent set will be considered for optimisation and each CUDA thread will consider at most one vertex. Because inter-block synchronisation is not possible, the host-side code waits for all kernel launches to return before proceeding to the next independent set. Relaunching a kernel may involve some overhead but CUDA threads are lightweight, so the effect of this overhead is mitigated.

Algorithm 6 Colouring of a vertex with ID = VERTEX_INDEX using the *First Fit* algorithm

```
int color = INITIAL_COLOR_TO_TRY;
const set<vtkIdType> & adjSet = vertexAdjacency->at(VERTEX_INDEX);

set<vtkIdType>::const_iterator it = adjSet.begin();
while(it != adjSet.end())
{
    if(colorOfVertex[*it] == color)
    {
        it = adjSet.begin();
        color++;
        continue;
    }
    it++;
}

colorOfVertex[VERTEX_INDEX] = color;
```

Algorithm 7 Colouring the entire mesh using the *First Fit* algorithm. COLOR_VERTEX is the function described in Algorithm 6

```
vector< set<vertexID> > * vertexAdjacencyList = mesh->getVertexAdjacencyList();
int nVertices = vertexAdjacencyList->size();

int maxColor = -1;
int * colorOfVertex = new int[nVertices];
for(int index = 0; index < nVertices; index++)
    colorOfVertex[index] = -1;

for(int vertexIndex = 0; vertexIndex < nVertices; vertexIndex++)
    COLOR_VERTEX(INITIAL_COLOR_TO_TRY = 0, VERTEX_INDEX = vertexIndex);
```

Algorithm 8 The optimisation method *OptimisationAlgorithm::optimize(...)* using OpenMP.

```
for(int independentSetNo = 0; independentSetNo < numberOfSets; independentSetNo++)
{
int setIterator;
vertexID iSet[] = independentSets[independentSetNo];

#pragma omp parallel for private(setIterator)
    for(setIterator = 0; setIterator < verticesInSet; setIterator++)
    {
        cavityID cavity = iSet[setIterator];
        Vector2d newCoords;

        if(!meshCavities[cavity].isOnBoundary())
            newCoords = relocateInnerVertex(...);
        else
            newCoords = relocateOuterVertex(...);
    }
}
```

Algorithm 9 The optimisation method *OptimisationAlgorithm::cudaOptimize(...)*.

```
for(int independentSetNo = 0; independentSetNo < numberOfSets; independentSetNo++)
{
    dim3 numBlocks(ceil((double) verticesInSet / threadsPerBlock));
    kernel<<<numBlocks, threadsPerBlock>>>(independentSets[independentSetNo]);
    cudaThreadSynchronize();
}

__device__ void kernel(IndependentSet iSet)
{
    int vertex = blockIdx.x * blockDim.x + threadIdx.x;
    if(vertex < verticesInSet)
    {
        cavityID cavity = iSet[vertex];
        Vector2d newCoords;

        if(!meshCavities[cavity].isOnBoundary())
            newCoords = relocateInnerVertex(...);
        else
            newCoords = relocateOuterVertex(...);
    }
}
```

4.5 Optimisation techniques

Having described the implementation of *CUDAMesh64*, the data structures used and the way parallel execution takes place, we can now assemble a list of optimisation techniques which assist in achieving higher performance from a CUDA device. Some of these techniques also offer some speedup on the host side; however, the emphasis here is given to the benefits from the device side. These optimisations can be turned on or off by setting the appropriate definitions in the *Configuration* header file.

4.5.1 Treating the metric tensor field as graphics texture

As was described in 4.3.8, the metric tensor field has been implemented both as a continuous function which returns the magnitude of the metric tensor in a requested point or as a matrix of metric samples (discrete form). The latter approach is the one that makes sense for real problems, since the error metric is almost always given as a collection of node-wise discretised samples (whereas the continuous version can only be used for demonstration purposes).

Having a matrix of metric samples, there are two ways it can be used. The naïve approach is to store this matrix in global memory. On the host side, this is the only available option. On the device side, however, this matrix can be stored in *Texture Memory*, offering the benefits described in 3.1.1. Apart from that, interpolating metric values is done by dedicated hardware outside CUDA cores, so execution resources are relieved from all time-consuming calculations that are needed if the metric is stored in global memory, as can be seen in Algorithm 10. The only thing that has to be executed when using texture memory is simply a statement like:

$$float4\ metric = tex2D(metricTexture, iCoordinate, jCoordinate);$$

It should be noted, however, that CUDA limits the variety of datatypes that can be used in texture memory. As far as floating-point values are concerned, only 1-, 2- and 4- element vectors of single-precision floats are permitted. Representing the metric tensor field using single-precision is not really a problem in terms of accuracy. The only problem is that it could be claimed that we are unfair when we compare execution performance between CPU and GPU, loading the CPU with double the amount of data, i.e. double the memory traffic and double the cache load. On the other side, single-precision values are converted back to double-precision ones once they are fetched into the core, so these extra conversion instructions make up somehow for the lower bandwidth and lower cache usage "cheat".

Algorithm 10 Accessing the discrete metric tensor field from host code or from device code in case the field is stored in global memory.

```
double iIndex = jCoordinate * metricDim;
int i = floor(((metricDim - 1) / metricDim) * iIndex);
iIndex -= i;

double jIndex = iCoordinate * metricDim;
int j = floor(((metricDim - 1) / metricDim) * jIndex);
jIndex -= j;

if(i == (metricDim - 1)
    metric = metricValues[metricDim * (metricDim - 1) +   j  ] * (1 - jIndex) +
             metricValues[metricDim * (metricDim - 1) + (j+1)] *    jIndex;
else if(j == metricDim - 1)
    metric = metricValues[(i+1) * metricDim - 1] * (1 - iIndex) +
             metricValues[(i+2) * metricDim - 1] *    iIndex;
else
    metric = metricValues[  i   * metricDim +   j  ] * (1 - iIndex) * (1 - jIndex) +
             metricValues[  i   * metricDim + (j+1)] * (1 - iIndex) *    jIndex    +
             metricValues[(i+1) * metricDim +   j  ] *    iIndex    * (1 - jIndex) +
             metricValues[(i+1) * metricDim + (j+1)] *    iIndex    *    jIndex;
```

4.5.2 Putting boundary vertices in dedicated independent sets

In 3.1.2 it is mentioned that thread divergence can seriously degrade execution performance. As can be seen in Algorithm 9, boundary vertices are smoothed using a different method than inner vertices. This can cause thread divergence in case two threads of the same warp are assigned one inner and one boundary vertex at the same time. In order to avoid this situation, boundary vertices can be coloured so that they belong to dedicated independent sets. This way, all threads within a warp will be assigned only inner or only boundary vertices at any given time.

Colouring using dedicated sets for boundary vertices can be done as described in Algorithm 11. Inner vertices are coloured exactly as was described in Algorithm 7. After doing so, boundary vertices are coloured starting with a fresh colour instead of the first colour, i.e. creating a new independent set. This way, all boundary vertices will be put into dedicated independent sets.

4.5.3 Stripping *Cavity* objects off *Element* information

As was described in 4.3.7, *Cavities* include the IDs of all surrounding vertices and all elements defining the cavity. Element information is stored in a cavity for the purpose of evaluating the objective

Algorithm 11 Colouring the entire mesh using the *First Fit* algorithm, putting boundary vertices to dedicated independent sets. COLOR_VERTEX is the function described in Algorithm 6

```
vector< set<vertexID> > * vertexAdjacencyList = mesh->getVertexAdjacencyList();
int nVertices = vertexAdjacencyList->size();

int maxColor = -1;
int * colorOfVertex = new int[nVertices];
for(int index = 0; index < nVertices; index++)
    colorOfVertex[index] = -1;

for(int vertexIndex = 0; vertexIndex < nVertices; vertexIndex++)
    if(!mesh->getCavities()[vertexIndex].isOnBoundary())
        COLOR_VERTEX(INITIAL_COLOR_TO_TRY = 0, VERTEX_INDEX = vertexIndex);

int minOuterColor = maxColor + 1;

for(int vertexIndex = 0; vertexIndex < nVertices; vertexIndex++)
    if(mesh->getCavities()[vertexIndex].isOnBoundary())
        COLOR_VERTEX(INITIAL_COLOR_TO_TRY = minOuterColor,
                     VERTEX_INDEX = vertexIndex);
```

functional (4.3.9). In 4.3.10, however, it was explained that the optimisation algorithm by Freitag et al. was not possible to be implemented. The algorithm by Pain et al., on the other hand, does not use any element information; therefore, this piece of data can be omitted from the *Cavity* data structure, saving both memory space and bandwidth, as well as increasing (possibly) cache hit rates. This optimisation can have a performance benefit both for host and device codes.

4.5.4 Using on-chip memory primarily as L1 cache

In devices of compute capability 2.0 and above, the same on-chip memory is used both as shared memory and L1 cache. Two configurations are possible [nC10a]:

- Configured as 48KB of shared memory with 16KB of L1 cache (which is the default)
- Configured as 16KB of shared memory with 48KB of L1 cache

The unstructured nature of anisotropic mesh adaptivity has not allowed us to use shared memory explicitly. On the other hand, a hardware-managed L1 cache exploits data locality in a much better way. Choosing the second configuration can be done by preceding the kernel invocation with a statement like:

$$cudaFuncSetCacheConfig(optimizationKernel, cudaFuncCachePreferL1);$$

4.6 Implementation difficulties

This section lists various difficulties that were encountered during the development of *CUDAMesh64*. These problems either prevented us from implementing more features for the application or made us run out of time for the submission of this project.

4.6.1 Automatic Differentiation

As was described in 2.1.7, the algorithm by Freitag et al. uses objective functionals and their gradients in order to optimise a mesh cavity. The gradient of a functional, as mentioned in 4.3.9, can be computed using an *Automatic Differentiation* tool. Unfortunately, all AD tools we were able to find are classified into two categories, none of which is suitable for *CUDAMesh64*.

The first category includes AD tools, like OpenAD [MoANL], that parse the C source code implementing the function-to-be-differentiated and output new C source code implementing the gradient of this function. This way, we can obtain the C source code of the gradient and compile it for CUDA. Unfortunately, tools in this category can work only with C code, while C++ is only partially or not at all supported. The *ObjectiveFunctional::evaluate(...)* method relies upon *Cavity* and *Element* objects, so it implemented using (unsupported by AD) C++ code. By the time we realised that we would need an AD tool, it was too late to convert our codebase to pure C.

The second category contains tools and libraries that are called from a program, passing as argument a pointer to the function-to-be-differentiated in runtime. It is obvious that this approach cannot be used on CUDA, since these tools do not produce any C source code. Moreover, the libraries are compiled for conventional architectures, which means that the tool itself cannot be used from CUDA code; it can only run on the host. The only way we could use such an AD tool on CUDA would be taking its source code and porting it to the GPU, a task that is out of question for a time-constrained project like *CUDAMesh64*.

4.6.2 CUDA linker

A fact that is not documented in nVIDIA's CUDA Programming Guide [nC10a] regards some linker restrictions. Current CUDA hardware cannot call functions; therefore, everything has to be inlined in the kernel. This well-known property does not explain why the CUDA linker refuses to link into the same executable two functions that have been compiled to different object files. This restriction makes impossible, for example, using *Vector2d* overloaded operators inside the optimisation algorithm, because *Vector2d* and *OptimizationAlgorithm* are compiled to two different object files, Vector2d.o and OptimizationAlgorithm.o.

The only solution we could come up with is including all source code in a single file. *CUDAMesh64* consists of only one source file, "application.cu", which includes all necessary header files, as any regular C++ program would do. Each header file, however, includes in turn the .inl file which implements the class / structure / namespace. In other words, instead of including header files on top of implementation files, we do exactly the opposite. This way, after the C++ pre-processing stage, the compiler has just one big source file as input, so it builds only one object file as output.

4.6.3 METIS and two-level graph colouring

In an effort to exploit data locality as much as possible, a two-level colouring scheme was considered. In this scheme, the whole mesh would be partitioned into many small partitions, each one consisting of as few as 5 or 6 vertices, and the colouring algorithm would extract *independent sets of partitions*. Each CUDA thread would be assigned one partition, therefore optimising adjacent cavities in a serial way (no hazard of incorrect execution) and making better use of on-chip memory (either as software-controlled shared memory or hardware-managed L1 cache). Since all partitions belonging to the same independent set would be independent from each other, all threads could run in parallel and still output a correct result. Mesh partitioning can be done using a classic tool, like *METIS* [Lab].

Although two-level colouring is an interesting optimisation to explore in order to improve memory access latency, it was not implemented at this point because profiling the application indicated more serious performance bottlenecks elsewhere. Another interesting question that arises is about the complexity of two-level colouring. It can be shown that this optimisation is followed by some overhead work, which takes more than linear time (maybe even quadratic) to be executed. The question that remains to be answered is whether this overhead is worthwhile, taking into account that the *Anisotropic Mesh Adaptivity* problem has a complexity of $O(n^2)$, as will be shown in Chapter 5.

Now that the essential algorithmic and architectural background has been analysed and all design and implementation details have been described, we can proceed to the evaluation of *CUDAMesh64*'s performance, where it is shown how well this application is executed on CUDA in comparison to Intel's Xeon processors, how each optimisation affects performance and what the main bottlenecks are in this version of *CUDAMesh64*.

Chapter 5

Experimental results and evaluation

This chapter presents experimental results regarding the performance of *CUDAMesh64*, comparing absolute times and speedups between CPU serial, CPU mult-threaded and CUDA versions, exploring the right execution configuration with the help of the *CUDA Occupancy Calculator*, attempting to spot the main performance bottlenecks along with suggestions on how to overcome them in future releases and estimating the algorithmic complexity of the *Anisotropic Mesh Adaptivity* problem.

All experiments took place on a workstation which hosts two Intel "Clovertown" quad-core Xeon X5355 CPUs (2.66GHz) and is equipped with 4GB of main memory and a Tesla C2050 graphics board. The operating system at the time of experimentation was Ubuntu Server running Linux kernel 2.6.32-24-server x86_64. CPU code was compiled with GCC version 4.4.3, whereas for GPU code we used CUDA SDK 3.1 and CUDA compilation tools, release 3.1, V0.2.1221. Experiments were done using nVIDIA Forceware driver, version 256.40.

Every measurement presented in this chapter is the result of repeating the same experiment 5 times and taking the average value (apart from serial CPU execution, which would need whole days to complete). Additionally, CUDA times include the time it takes to copy data between host and device, but no measurement includes the time it takes to read in the unstructured grid, construct the mesh, colour it or write back the results to the output VTK file. In essence, the time to copy data between host and device is not important because these transfers take place only twice during an execution (copying the initial mesh to the device at the beginning and copying the adapted mesh back to the host at the end) and when we have thousands of iterations this time is amortised.

5.1 Execution Configuration

The very first thing that has to be done before taking any measurements is to find out the best execution configuration. nVIDIA provides a tool called *CUDA Occupancy Calculator* [nC], which gives valuable information about the warp occupancy of a CUDA multiprocessor. This occupancy is directly related with the peak performance which can be expected from a CUDA kernel. The highest possible occupancy in *Fermi* devices is 48 warps per multiprocessor.

Compiling *CUDAMesh64* with the additional flag --ptxas-options=-v instructs the compiler to output details about register usage by the CUDA kernel. Depending on which implementation of the *Metric* class we use, register usage is as shown in Table 5.1. Figure 5.1 shows the multiprocessor warp occupancy for the first three cases (59-62 registers per kernel) and Figure 5.2 depicts the occupancy for the discrete-textured case. As can be seen in these diagrams, the CUDA kernel uses too many registers in all cases:

Metric Version	Register Count
Linear	62
Sinusoidal	59
Discrete non-textured	59
Discrete textured	51

Table 5.1: Register usage by the CUDA kernel, depending on which version of *Metric* is used.

- With a register count of 59-62, maximum performance is achieved when using 48, 64, 112, 128, 240, 256, 496 or 512 threads per block and this performance is only $\frac{16}{48} = 33.3\%$ of the potential peak.
- With a register count of 51, maximum performance is achieved when using 80, 96, 176, 192, 272 or 288 threads per block and this performance is only $\frac{18}{48} = 37.5\%$ of the potential peak.

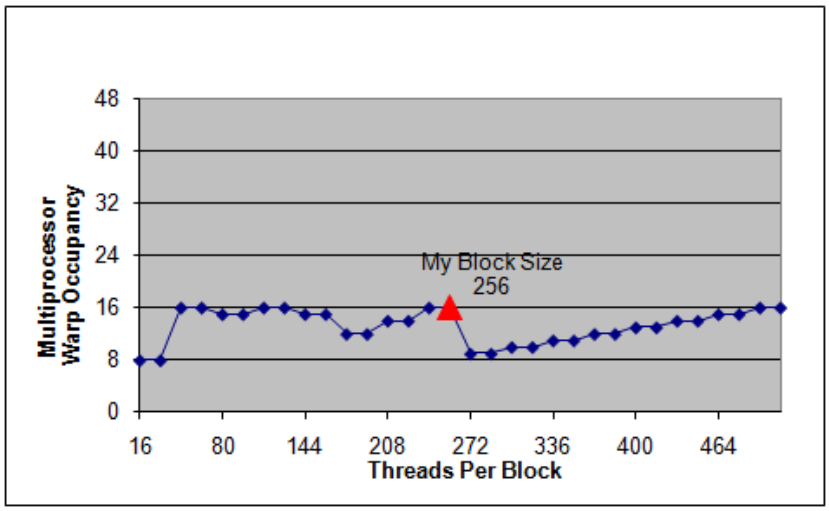

Figure 5.1: Multiprocessor warp occupancy when the kernel uses 59-62 registers.

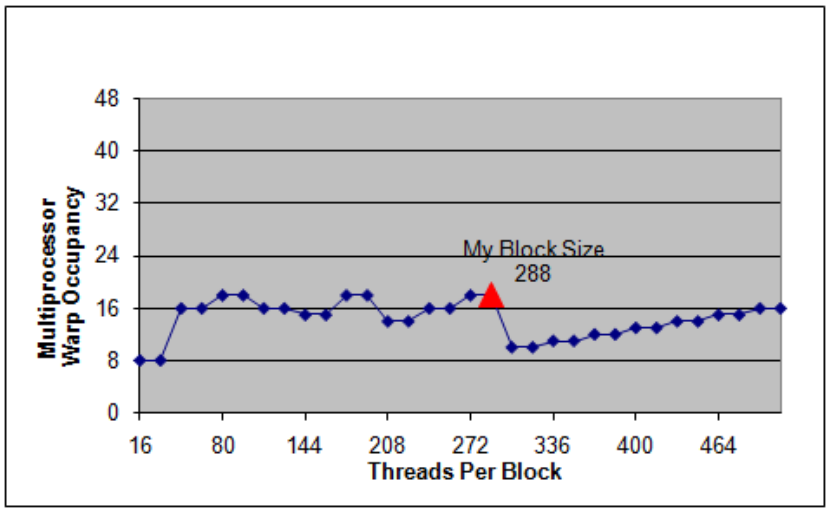

Figure 5.2: Multiprocessor warp occupancy when the kernel uses 51 registers.

Comparing the occupancy diagrams, it is seen that using texture memory for the metric tensor field is expected to offer even more benefits than just the ones described in Chapter 3. It should be noted,

Mesh Size	Vertices	Iterations	Time [sec]	Normalised Time [nsec]
10×10	91	10,000	2.33	2560.44
100×100	6417	10,000	3.56	55.48
200×200	25472	10,000	4.88	19.16
300×300	56878	10,000	8.46	14.87
500×500	157673	10,000	19.39	12.30
800×800	402849	10,000	48.22	11.97
1000×1000	627973	10,000	75.00	11.94
2000×2000	2512380	10,000	297.72	11.85
3000×3000	5654659	10,000	665.85	11.78

Table 5.2: Time results for various mesh sizes, using the basic CUDA version of *CUDAMesh64* (i.e. no optimisations turned on) and the continuous *Sinusoidal Metric*.

however, that occupancy diagrams are not necessarily indicative of the actual performance. The occupancy is calculated just by the number of registers used by a kernel. There are also other factors which affect performance, e.g. memory access latency and data locality.

5.2 Scaling with different mesh sizes

In this section we try to measure CUDA performance using various mesh sizes. CUDA is a massively parallel and floating-point capable architecture, which implies that good performance is expected to be observed in really large problems. In order to prove this claim, we ran a series of measurements using mesh sizes from 10×10 up to $3,000 \times 3,000$ (larger meshes would not fit in the available main memory of the workstation). These measurements correspond to the basic CUDA version of *CUDAMesh64*, i.e. no optimisations were turned on, using the continuous *Sinusoidal* metric.

Experimental results are tabulated in Table 5.2, where *Time* is the total execution time for 10,000 iterations and *Normalised Time* is the time in nanoseconds per cavity per iteration. Figure 5.3 presents these results in a graphical way. As can be seen, there is not enough parallelism in small meshes, so the GPU cannot be exploited to all its extent and the overhead of copying mesh data to the device and launching CUDA kernels is not justified. As a general rule, it can be said that CUDA execution starts becoming meaningful for mesh sizes above 200×200.

5.3 Basic CUDA version speedup

The next thing to investigate is how the basic CUDA version compares to serial and OpenMP CPU versions. The workstation we are working on allows us to run OpenMP applications using up to 8 threads (recall that this machine is equipped with two quad-core, non-HyperThreaded Xeons). Before proceeding to CPU-GPU comparison, we have to find out which exact execution configuration yields the best CUDA performance. According to the occupancy diagrams, maximum occupancy is achieved when using 48, 64, 112, 128, 240, 256, 496 or 512 warps per CUDA block. Table 5.3 and Figure 5.4 show the execution time achieved by each configuration, when running on a $2,000 \times 2,000$ mesh ($10,000$ iterations). Note that, from now on, we measure performance using only the discretised form of the metric tensor field, because this is the form used in a real adaptivity problem.

The best configuration seems to be 32 threads per CUDA block. Although multiprocessor occupancy is lower using 32 threads per block than using 48 and above, which in theory would imply lower

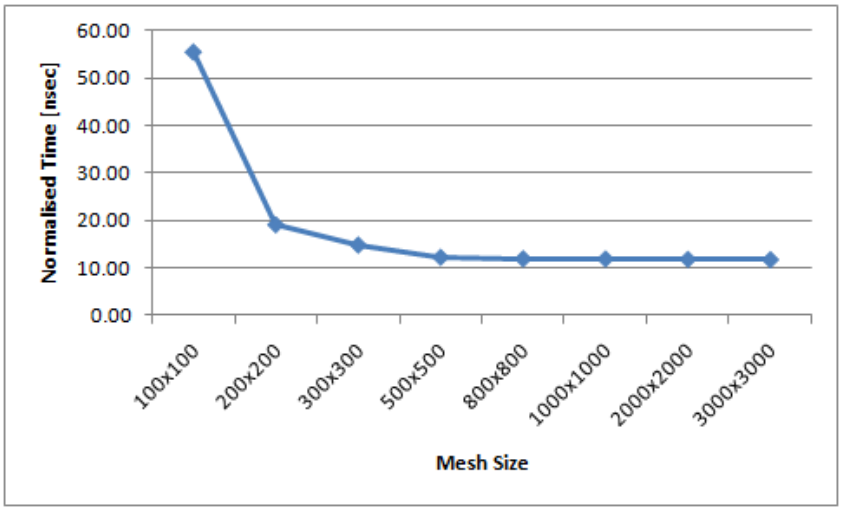

Figure 5.3: Time per vertex per iteration for various mesh sizes, using the basic CUDA version and the continuous *Sinusoidal Metric*.

Threads per Block	Time [sec]	Normalised Time [nsec]
16	981.9	39.08
32	820.16	32.64
48	889.58	35.41
64	1,039.23	41.37
80	931.07	37.06
96	988.02	39.33
112	954.45	37.99
128	1,037.19	41.28

Table 5.3: Execution time of the basic CUDA version on a $2,000 \times 2,000$ mesh (10,000 iterations) using various execution configurations.

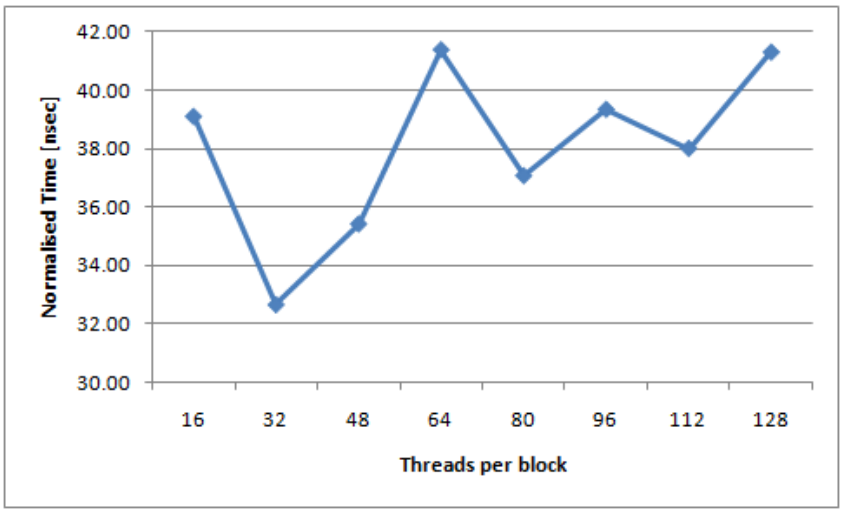

Figure 5.4: Normalised execution time of the basic CUDA version on a $2,000 \times 2,000$ mesh using various execution configurations

Mesh Size	Serial [sec]	OpenMP [sec]	CUDA [sec]
500 × 500	2,512.33	544.11	51.95
800 × 800	7,770.02	1,931.72	131.40
1000 × 1000	11,797.94	2,902.62	203.22
2000 × 2000	55,627.80	12,997.13	819.58

Table 5.4: Performance comparison for various mesh sizes, using the basic CUDA version.

performance, the fact that using less threads per block yields better performance indicates there is a bottleneck in memory access. Having less threads per block means that we have more blocks running on each multiprocessor and, in turn, having more blocks per multiprocessor helps us hide memory latency (whereas having more threads per block helps us hide instruction latency). Another reason behind this oddity is the fact the number of vertices in every independent set is not a multiple of the size of the CUDA block. This means that the block processing the last vertices in a set contains threads that remain idle. The larger the block size, the more likely it is that a lot of threads in these last blocks will have no cavities to optimise.

Using the optimal execution configuration we ran a series of tests to compare the basic CUDA version to the serial and the eight-threaded OpenMP CPU versions. Using mesh sizes up to $2,000 \times 2,000$ and performing 10,000 iterations in each case, we measured the absolute execution time of each version and the relative speedup between serial and OpenMP, between serial and CUDA and between OpenMP and CUDA. Table 5.4 shows timing results. Figure 5.5 shows the relative speedup between the three versions of *CUDAMesh64*. The 8-threaded OpenMP version is 4-5 times faster than the serial code, whereas engaging the GPU offers a speedup of 48-68 times over the serial code and 10-16 times over OpenMP.

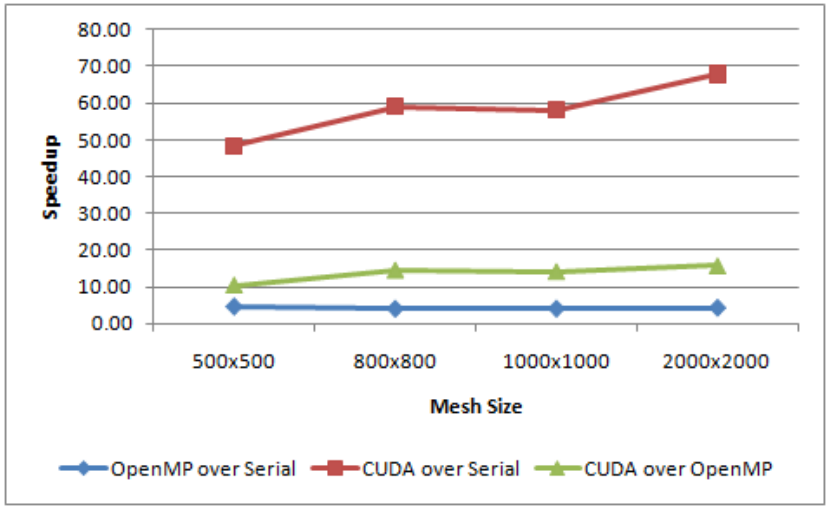

Figure 5.5: Speedup between serial, OpenMP and basic CUDA versions.

Before proceeding to the optimisations section, we deem as necessary to test the performance of *CUDAMesh64* when the *Cavity* class is defined without element information. The reason we believe that this optimisation belongs to this section is that stripping cavities off element information affects both CPU and GPU executions. Table 5.5 and Figure 5.6 show the experimental results. Absolute execution time has been reduced in all three versions of *CUDAMesh64*; however, the relative speedup remains pretty much the same.

Mesh Size	Serial [sec]	OpenMP [sec]	CUDA [sec]
500 × 500	2,242.52	510.19	51.06
800 × 800	7,099.94	1,812.79	128.92
1000 × 1000	10,797.34	2,793.12	199.63
2000 × 2000	54,588.46	12,864.17	805.59

Table 5.5: Performance comparison for various mesh sizes, using element-less cavities and the basic CUDA version.

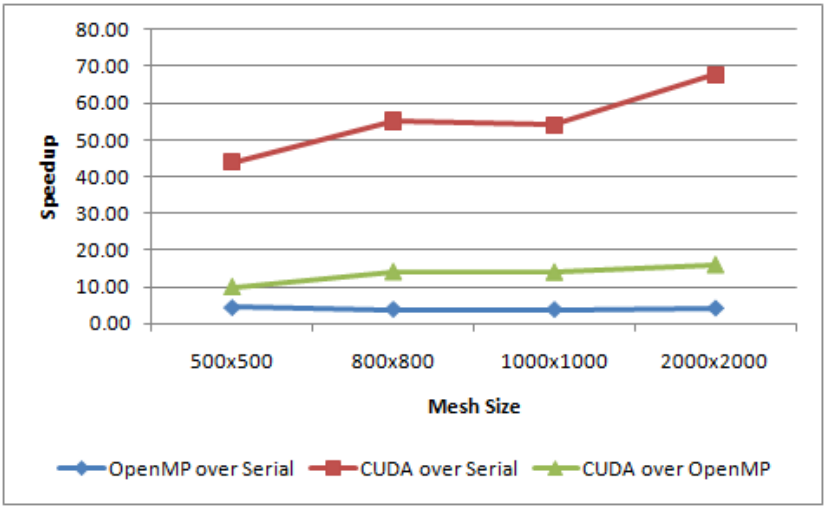

Figure 5.6: Speedup between serial, OpenMP and basic CUDA versions using element-less cavities.

5.4 Optimisations

Having seen how well the basic CUDA version competes against the OpenMP one, it is interesting to see how performance can benefit from the optimisation described in Section 4.5 and how big the CPU-GPU gap can become. All experiments below have been done using the element-less version of the *Cavity* class.

The major optimisation, and the one our experimentation starts with, is the usage of texture memory to store the metric tensor field. It has been already mentioned that this optimisation results in a CUDA kernel which uses less registers than the basic version; therefore, it is necessary to repeat the best configuration test. The results can be seen in Table 5.6 and Figure 5.7. Once again, although the theoretically best configuration should be one of the configurations indicated by Figure 5.2, in practice 16 threads per block seems to execute at the fastest speed.

An additional point worth mentioning is that, by comparing execution times for the $2,000 \times 2,000$ mesh between Table 5.2 and Table 5.6, we can see that engaging texture memory almost takes away the overhead of using the discrete form of the metric tensor field.

Using the optimal execution configuration we ran a series of tests to assess the performance of the texture-memory version, as well as how much this version can benefit from using the on-chip memory as L1 cache and putting boundary vertices to dedicated independent sets. Using mesh sizes up to $2,000 \times 2,000$ and performing 10,000 iterations in each case, we measured the absolute CUDA execution time, enabling one more optimisation in each successive experiment. Timing results can be seen in Table 5.7.

Threads per Block	Time [sec]	Normalised Time [nsec]
16	323.05	12.86
32	360.57	14.35
48	367.14	14.61
64	385.02	15.32
80	380.73	15.15
96	385.23	15.33
112	378.59	15.07
128	387.55	15.43

Table 5.6: Execution time of the texture-memory CUDA version on a $2,000 \times 2,000$ mesh (10,000 iterations) using various execution configurations.

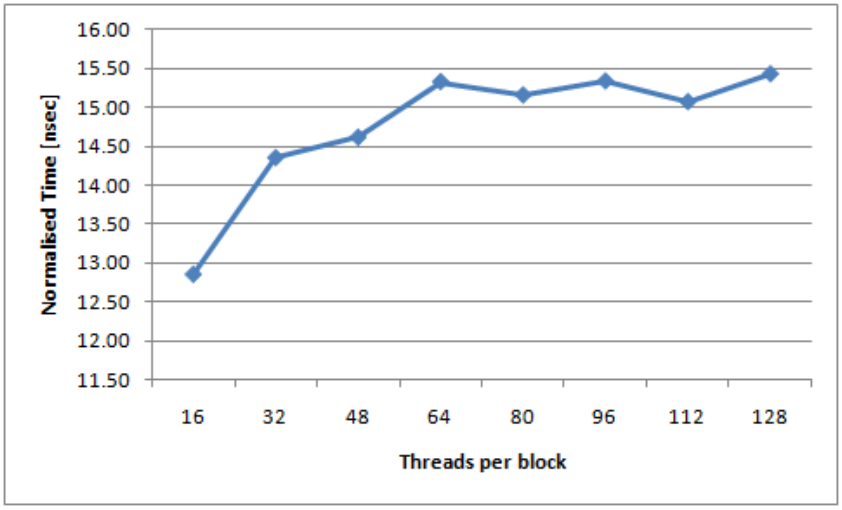

Figure 5.7: Normalised execution time of the texture-memory CUDA version on a $2,000 \times 2,000$ mesh using various execution configurations

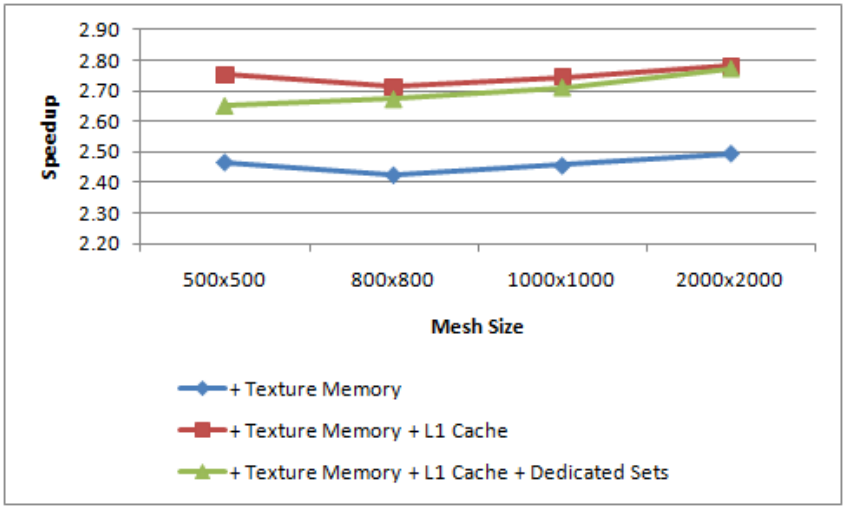

Figure 5.8: Speedup between the three CUDA optimisations.

Figure 5.8 shows the relative speedup between the three CUDA optimisations. Using texture memory for the metric tensor field offers a performance boost of $\times 2.5$. Enabling the on-chip memory to operate

Mesh Size	Serial [sec]	OpenMP [sec]	CUDA [sec]	+ TM [sec]	+ L1 [sec]	+ DS [sec]
500 × 500	2,242.52	510.19	51.06	20.70	18.55	19.26
800 × 800	7,099.94	1,812.79	128.92	53.15	47.52	48.23
1000 × 1000	10,797.34	2,793.12	199.63	81.28	72.72	73.64
2000 × 2000	54,588.46	12,864.17	805.59	322.87	289.71	290.65

Table 5.7: Performance comparison for various mesh sizes, enabling CUDA optimisations; fifth column: CUDA + texture memory; sixth column: CUDA + texture memory + on-chip memory as L1 cache; seventh column: CUDA + texture memory + on-chip memory as L1 cache + dedicated sets for boundary vertices.

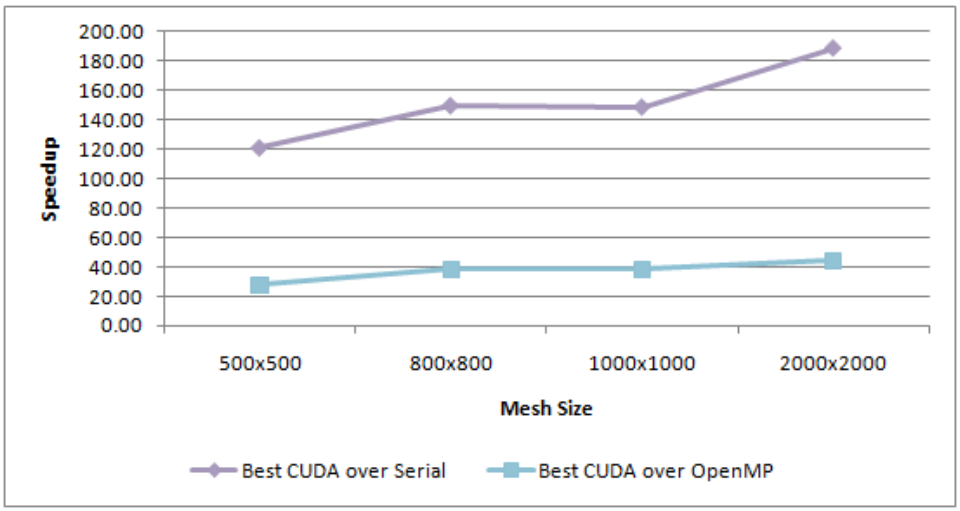

Figure 5.9: Speedup between the best CUDA version and the serial and OpenMP CPU versions.

as a 48KB L1 cache further boosts performance, bringing it to a level of the basic version ×2.75. Oddly enough, the dedicated-sets optimisation does not assist in reducing execution time; on the contrary, timing results are slightly worse. This unexpected behaviour indicates two things:

(a) Having applied all other optimisations, memory latency is now more significant. Thread divergence does not affect performance, since threads would stall anyway waiting for data to be fetched into the multiprocessor.

(b) Independent sets containing boundary vertices are too small. This means that when it comes to boundary vertex smoothing, it is not possible to exploit the hardware to all its extent. As an example, colouring the $2,000 \times 2,000$ mesh results to the following independent sets: Set0(673,510 vertices), Set1(539,053 vertices), Set2(601,044 vertices), Set3(523,783 vertices), Set4(161,806 vertices), Set5(5,183 vertices), Set6(1 vertex), Set7(4,000 vertices), Set8(3,996 vertices), Set9(4 vertices). The difference between "inner" and "boundary" independent sets is obvious.

Figure 5.9 shows the relative speedup between the best CUDA version and CPU versions. Taking the dedicated sets oddity into consideration, the best CUDA version is the one using texture memory for the metric tensor field and on-chip memory as L1 cache. Compared to the serial code, the best CUDA code executes 120-190 times faster, as mesh sizes grow. The speedup over the OpenMP version is also significant, lying in the $\times 27 - \times 45$ range.

5.5 Complexity of the Anisotropic Mesh Adaptivity problem

The final thing we want to conclude out of this project is the complexity of the *Anisotropic Mesh Adaptivity* problem. The complexity is defined with respect to the number of mesh vertices n and can be found by determining the number of iterations that have to be performed until we converge to an adapted mesh. In this test we have used two medium-sized meshes, a 100×100 and a 200×200 one. The results of gradually adapting these meshes can be seen in Figure 5.10 and Figure 5.11, respectively.

Looking at the results, we can say that the 100×100 mesh, which consists of $6,417$ vertices, needs $10,000 - 20,000$ iterations to completely adapt to the error metric. Similarly, the 200×200 mesh, which consists of $25,472$ vertices, needs $40,000 - 60,000$ iterations. This observation leads to the conclusion that, in general, a mesh consisting of n vertices needs $\Theta(n)$ iterations to adapt. Consequently, the complexity of the *Anisotropic Mesh Adaptivity* problem using the optimisation algorithm by Pain et al. is $\Theta(n^2)$.

Another interesting observation is the reason why it takes so long for a mesh to adapt. Anisotropic problems require larger-scale vertex relocation, i.e. a vertex may be moved several "pixels" away from its original position p_i. Even if we knew the exact final position p_f for that vertex from the very beginning, we would have to wait for several iterations until all vertices between p_i and p_f be also relocated; we cannot relocate the vertex under consideration in one step because if we do so we may invert cavities and invalidate the mesh.

Recapitulating, this chapter presented the results from our experimentation with *CUDAMesh64*. It was shown that the CUDA version can offer performance gains up to $\times 190$ compared to the simple, serial CPU code and $\times 45$ over the eight-threaded OpenMP version. Additionally, we found out that the theoretically best execution configuration does not necessarily yield the best timing results in the presence of other important obstacles, like memory latency. Finally, it was estimated that the algorithmic complexity of the *Anisotropic Mesh Adaptivity* problem is quadratic with respect to the number of mesh vertices. The next, and final, chapter of this report summarises all these points and proposes topics that are left as future work.

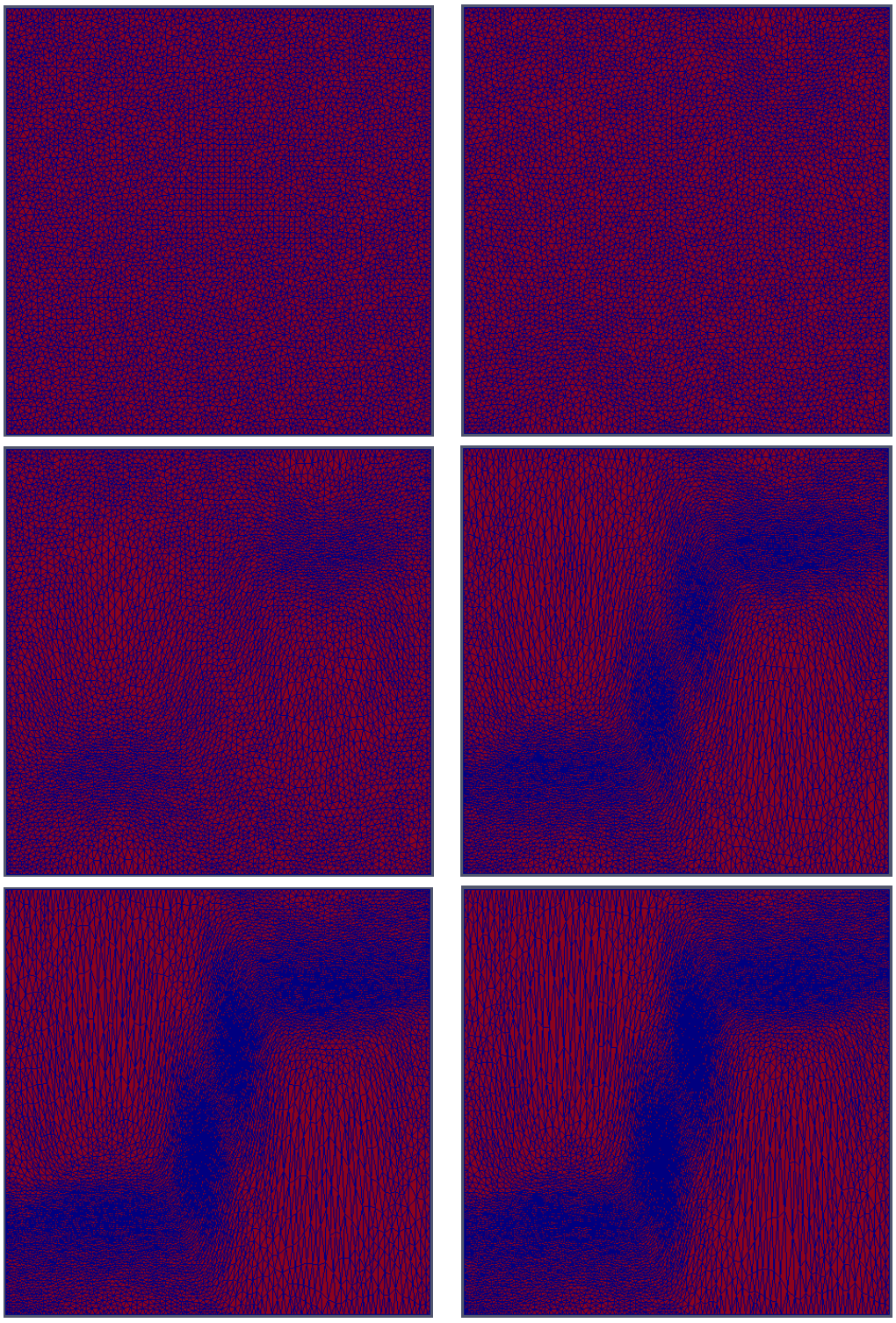

Figure 5.10: Gradually adapting a 100 × 100 mesh. Top line: The original mesh (left), after 100 iterations (right); Middle line: After 1,000 iterations (left), after 5,000 iterations (right); Bottom line: After 10,000 iterations (left), after 20,000 iterations (right).

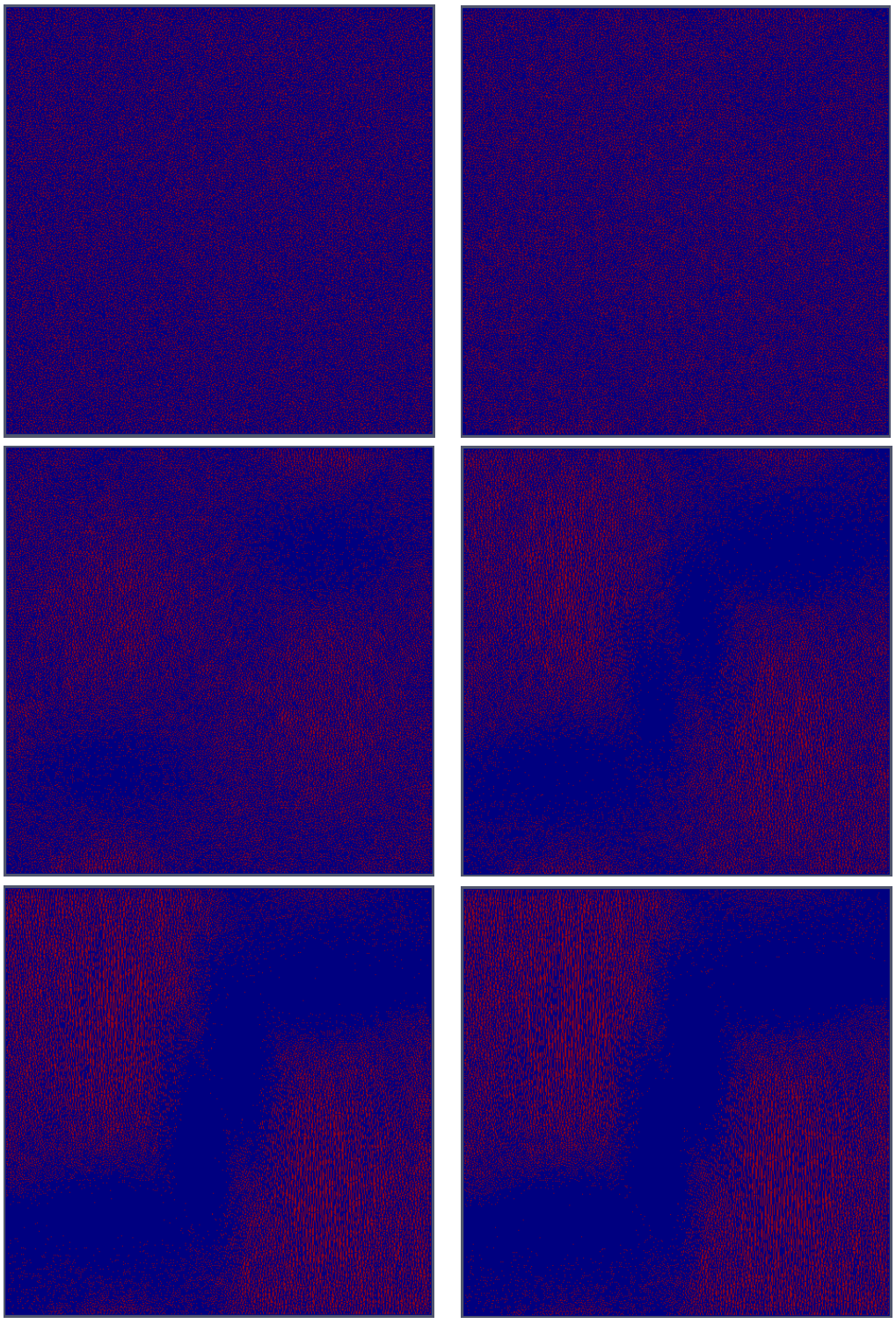

Figure 5.11: Gradually adapting a 200 × 200 mesh. Top line: The original mesh (left), after 200 iterations (right); Middle line: After 4,000 iterations (left), after 16,000 iterations (right); Bottom line: After 40,000 iterations (left), after 60,000 iterations (right).

Chapter 6

Conclusions and future work

This last chapter constitutes a recapitulation of what has been done throughout this project, the main algorithmic concepts encountered, the key architectural points of CUDA, the benefits this architecture can offer in terms of performance compared to conventional hardware approaches and the experience we gained through our avocation with this project. Additionally, we summarise the main performance bottlenecks of *CUDAMesh64* and propose ways in which these problems can be mitigated, along with topics that remain open to further study and future investigation. Finally, we attempt to link the knowledge acquired through this project to our previous experience with another high-performance architecture, IBM's *Cell Broadband Engine*, and explain why we think that programming in CUDA is much simpler and easier and still it results to greater speedups.

6.1 Conclusions

This project was proposed having in mind the creation of CUDA-enabled application, which was named *CUDAMesh64*, with the purpose of assessing this architecture with respect to a specific, applied mathematics problem.

6.1.1 CUDAMesh64

The basis of this project is the problem of *Anisotropic Mesh Adaptation*, i.e. the process of taking an unstructured input mesh and adapting it (or deforming it) as indicated by some metric. The algorithm we used to perform mesh adaptation is known as *Vertex Smoothing* and it has been implemented using the optimisation technique proposed by Pain et al. ([PUdOG01]). Another optimisation technique that was attempted is the one proposed by Freitag et al. ([FJP95]), which is based on optimising some objective functional, like the one proposed by Vasilevskii and Lipnikov ([VL99]). Unfortunately, there were technical difficulties which inhibited us from implementing this optimisation technique.

Correct Parallel Execution is a key point when a large scale application, like *CUDAMesh64*, is to be run on a parallel processing architecture. Freitag et al. have proposed a relative framework ([FJP98]) which can be easily implemented, ensures correctness of execution and leaves a lot of parallelism to be extracted out of a large problem (large mesh). This framework relies upon graph colouring, which means that we have to engage some colouring algorithm. It is clear that the amount of parallelism, therefore the expected execution performance, is affected in a critical way by the choice of a suitable colouring scheme. *First Fit Colouring*, a simple, greedy and serial colouring algorithm, was shown to be sufficiently good, although better schemes may help us achieve better performance.

nVIDIA's *Compute Unified Device Architecture* ([nC10a], [nC10b]) was the target platform of choice for our application. It is a massively parallel and floating-point capable architecture, ideal for solving complex scientific problems. The naïve CUDA version of *CUDAMesh64* can run up to ×68 faster than the serial CPU code and up to ×16 faster than an eight-threaded OpenMP code. Throughout our experiments with *CUDAMesh64* it was shown that a powerful feature of CUDA is the ability to use *Texture Memory* to store the metric tensor field. Doing so raises the relative speedup over serial and OpenMP versions, reaching a total speedup of up to ×190 and ×45, respectively. In fact, using texture memory can almost take away the overhead of using the discrete form of metric tensors.

Performance analysis also showed there are two serious bottlenecks. The most important seems to be the fact that the CUDA kernel occupies a lot of registers, which results in low multiprocessor occupancy. Even in the best version, i.e. the version occupying the fewest registers, register usage is still so high that it accounts for a theoretical 62.5% performance loss. This problem could be mitigated by breaking down the optimisation algorithm into many smaller stages and storing intermediate results; the output of one stage will be the input to the next one. Each stage will be executed by launching a dedicated CUDA kernel. The downside of this approach is that many more kernels will have to be launched compared to the current implementation of *CUDAMesh64*: instead of launching one kernel per mesh pass, if the algorithm is broken down into k parts, k kernels will have to be invoked for each pass. According to nVIDIA, however, kernels are very lightweight, so increasing the number of kernel invocations should not be a problem.

The second obstacle is memory access. Best performance is achieved when using few threads per block and not the values indicated by the occupancy calculator. The first thing that has to be done towards fixing memory latency is to improve data locality. Using the on-chip memory as a 48KB L1 cache is already proved to speed up execution. A two-level colouring scheme that takes locality into account could potentially improve execution speed even further. Although such a scheme was started in this project, there was not sufficient time to complete the work.

Another conclusion from this project regards the complexity of the *Anisotropic Mesh Adaptivity* problem when the algorithm by Pain et al. is used. Seeing how the mesh looks after various numbers of iterations, it is estimated that the complexity of this problem is $\Theta(n^2)$, n being the number of vertices in the mesh. Although other, more sophisticated algorithms, like the one by Freitag et al., are usually expected to converge faster to a solution, we highly doubt that this is really the case in anisotropic problems. The main reason why it takes so many iterations for a mesh to adapt does not seem to be algorithm accuracy or efficiency, but rather the fact that vertex relocation in anisotropic problems is of a much larger scale than in isotropic PDEs. Even if the algorithm can indicate the final position of a vertex precisely, this vertex cannot be relocated to that position in one step, since there are other vertices in between that have to be smoothed first.

6.1.2 nVIDIA's CUDA vs IBM's CBEA

Before starting this project we had some previous experience with another high-performance architecture, IBM's *Cell Broadband Engine* [IBM07], in the scope of numerical analysis project, where we studied the solution of the 2D advection PDE on the latter platform ([RPK+10]). Having spent time testing both architecture, we can comment on how easy it is to program on each platform and what one should expect in term of execution performance.

The first advantage of CUDA we noticed was from the programmer's perspective. When porting CPU code to the device, there are only a few modifications that have to be made. In contrast, porting existing code to CBE includes a complete rewrite so that new code is written specifically for the *Synergistic Processor Elements* (SPEs), the main vector execution cores of Cell. As a result, CUDA's approach is much more safe, as it is not very likely that new bugs will be introduced in the existing

codebase. Another advantage of the SIMT (Single Instruction Multiple Threads) concept is that parallelisation of an algorithm is done in a more automated way. There is no need for manual data vectorisation or explicit data transfer to a multiprocessor. When a multiprocessor needs some data, it access global memory directly, whereas in Cell these data have to be transferred explicitly to an SPE's *Local Storage* using *DMA Transfers*.

From an architectural point of view, CUDA can exploit much more parallelism, being able to manipulate thousands of threads. This way, it is easier to get higher performance even from naïve code. The key point is the organisation of threads in warps and the ability to switch between warps at no cost. In contrast, a typical CBE chip contains 8 SPEs, each one being able to run one context (thread) at a time. If, for any reason, a SPE thread stalls, there is no way to keep the SPE busy doing other useful stuff (apart from SPU context switching, the overhead of which is too high to be considered as a real alternative). Computation and communication can be overlapped in CBE as it is possible to have many outstanding DMA transfers; however, doing so requires some effort by the programmer and it is not always the solution to low performance (e.g. if a kernel needs a lot of data but executes only a few operations on them, overlapping computation-communication will not help much). The concept of warps and the ability to run hundreds of threads on each multiprocessor not only hides memory access latency but eliminates branch penalties as well.

On the other hand, Cell has its own points of excellence. Instead of relying on a relatively slow bus like PCI-Express, it engages a high-bandwidth *Element Interconnect Bus* (EIB), which can communicated directly with the main memory. EIB also lacks CUDA's restrictions on memory access patterns, like memory partitions and coalesced accesses. Moreover, each SPE is equipped with two heterogeneous pipelines, one being responsible for memory operations (IBM calls it "odd" pipeline) and the other executing floating-point operations (IBM calls it "even" pipeline). These pipelines operate independently from each other, so it is possible to avoid interruptions to the even pipeline if we make sure that the odd pipeline has fetched all necessary data into the register file before the even pipeline needs them.

From a programmer's perspective, CBE fully supports C++, which can result in easier development and less error-prone code. Furthermore, we encountered some undocumented aspects regarding CUDA, like the linker's incapability to link functions between different object files, which made us lose quite a lot of time trying to figure out why we could not get our code to compile. Finally, nVIDIA has not disclosed enough details about CUDA internals, like CUDA assembly language and register file specifications. Although manual instruction scheduling is a painful and time consuming task, having the full specifications of Cell enables us to optimise code to a much greater extent than what we are able to do on CUDA.

The general feeling we are left with is that CUDA resources do not seem to keep up with massive parallelism. For example, a multiprocessor can run 512 threads concurrently but it has only 64KB of on-chip memory (this is the amount of L1 cache in a modern CPU core supporting 2-way SMT). Another example is the significant performance penalty if a kernel occupies more than 20 registers; in Cell all 128 128-bit registers can be used without worrying about performance penalties. These restrictions are a legacy of CUDA's graphics processing ancestry. CUDA seems to be more suitable for very simple operations, i.e. small kernels, on very large amounts of data. This is why *"GPU is not CPU"*.

In the end, however, what matters the most is the result: CUDA can achieve much higher GFLOPS values. Still, we consider CBEA to be a much more sophisticated piece of engineering.

6.2 Future Work

There are various topics that remain open to further study. The most significant among them, and the ones to be investigated in the context of a publication following this project, are:

- Breakdown of the optimisation kernel, so that register usage is limited and multiprocessor occupancy is maximised.
- Graph colouring using a two-level scheme in an attempt to improve data locality and cache efficiency.
- Redoing the CPU experiments on a newer CPU, so that performance can be compared between hardware of the same "era".
- Finding out the individual contribution of the two issues of benefit from using texture memory for the metric tensor field, i.e. texture caching and hardware implementation of interpolation.

Taking proposals for future work a bit further, we would suggest that following topics are considered:

- Reattempt to implement the optimisation algorithm by Freitag et al., once an appropriate automatic differentiation tool is available or significant part of *CUDAMesh64* codebase can be rewritten in pure C.
- Porting *CUDAMesh64* to AMD's (formerly ATI) *Stream Architecture*, with the purpose of performing comparisons between these competing architectures.
- Rewriting the application in OpenCL so that there is an abstraction layer between the application itself and the underlying hardware.
- Using the port to OpenCL to experiment with AMD's OpenCL compiler, testing not only GPU code, but also the compiler's capabilities in vectorisation (SSE) and multi-core execution. With AMD's *Fusion* architecture of *Accelerated Processing Units* (APUs) being on the way, such an experimentation would be very interesting.
- Even experimenting with IBM's OpenCL compiler for Cell.
- Accelerating the 3D version of *Anisotropic Mesh Adaptivity* using GPGPU computing.

Bibliography

[AMC06] AMCG. Simple adapt example. http://amcg.ese.ic.ac.uk/index.php?title=Simple_Adapt_Example, April 2006.

[AOS06] Hussein Al-Omari and Khair Eddin Sabri. New graph coloring algorithms. *Journal of Mathematics and Statistics*, 2006.

[cfd08] Case Study: CFD. Technical report, Supercomputing 2008 CUDA Tutorial, November 17th 2008.

[cL99] François Labelle. Anisotropic triangular mesh generation based on refinement. http://www.eecs.berkeley.edu/ flab/cs294-5/project2/mesh.html, December 1999.

[cud08a] CUDA Basics. Technical report, Supercomputing 2008 CUDA Tutorial, November 17th 2008.

[cud08b] Introduction to CUDA. Technical report, Supercomputing 2008 CUDA Tutorial, November 17th 2008.

[cud08c] Optimizing CUDA. Technical report, Supercomputing 2008 CUDA Tutorial, November 17th 2008.

[ELD08] Erich Elsen, Patrick LeGresley, and Eric Darve. Large calculation of the flow over a hypersonic vehicle using a gpu. *Journal of Computational Physics*, 227(24):10148 – 10161, 2008.

[FJP95] Lori Freitag, Mark Jones, and Paul Plassmann. An Efficient Parallel Algorithm for Mesh Smoothing. In *INTERNATIONAL MESHING ROUNDTABLE*, pages 47–58, 1995.

[FJP98] Lori F. Freitag, Mark T. Jones, and Paul E. Plassmann. The Scalability Of Mesh Improvement Algorithms. In *IMA VOLUMES IN MATHEMATICS AND ITS APPLICATIONS*, pages 185–212. Springer-Verlag, 1998.

[Fre97] Lori A. Freitag. On Combining Laplacian And Optimization-Based Mesh Smoothing Techniques. In *TRENDS IN UNSTRUCTURED MESH GENERATION*, pages 37–43, 1997.

[GBT06] Dominik Göddeke, Christian Becker, and Stefan Turek. Integrating GPUs as fast coprocessors into the parallel FE package FEAST. In Matthias Becker and Helena Szczerbicka, editors, *19th Symposium Simulationstechnique (ASIM'06)*, Frontiers in Simulation, pages 277–282, September 2006.

[HB] Jared Hoberock and Nathan Bell. Thrust. http://code.google.com/p/thrust/.

[HB10] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.2.1.

[IBM07] IBM. Cell Broadband Engine Programming Tutorial. Technical report, 2007.

[Kita] Inc. Kitware. ParaView - Open Source Scientific Visualization. http://www.paraview.org/.

[Kitb] Inc. Kitware. The Visualization Toolkit. http://www.vtk.org/.

[Lab] Karypis Lab. METIS - Family of Multilevel Partitioning Algorithms. http://glaros.dtc.umn.edu/gkhome/views/metis.

[ML09] Applied Modelling and Computation Group Imerial College London. Fluidity. http://amcg.ese.ic.ac.uk/index.php?title=Fluidity, November 2009.

[MoANL] Mathemetics and Computer Science Division of Argonne National Laboratory. OpenAD. http://www.mcs.anl.gov/OpenAD/.

[nC] nVIDIA Corporation. CUDA GPU Occupancy Calculator. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[nC10a] nVIDIA Corporation. nVIDIA CUDA Programming Guide, Version 3.1. Technical report, 2010.

[nC10b] nVIDIA Corporation. nVIDIA CUDA Reference Manual, Version 3.1. Technical report, 2010.

[PFW+09] M. D. Piggott, P. E. Farrell, C. R. Wilson, G. J. Gorman, and C. C. Pain. Anisotropic mesh adaptivity for multi-scale ocean modelling. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1907):4591–4611, 2009.

[PUdOG01] C. C. Pain, A. P. Umpleby, C. R. E. de Oliveira, and A. J. H. Goddard. Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations. *Computer Methods in Applied Mechanics and Engineering*, 190(29-30):3771 – 3796, 2001.

[Rok10] Georgios Rokos. ISO Thesis: Study of Anisotropic Mesh Adaptivity and its Parallel Execution. *Imperial College London*, 2010.

[RPK+10] Georgios Rokos, Gerassimos Peteinatos, Georgia Kouveli, Georgios Goumas, Kornilios Kourtis, and Nectarios Koziris. Solving the advection PDE on the Cell Broadband Engine. In *Proceedings of the* 24^{th} *IEEE International Parallel & Distributed Processing Symposium*, 2010.

[TccS09] Julien Thibault and Inanç Şenocak. Incompressible Navier-Stokes Solver Implementation on Single, Dual and Quad GPU desktop platforms with CUDA™. http://coen.boisestate.edu/senocak/files/BSU_CUDA_Res_v5.pdf, 2009.

[VL99] Y. Vasilevskii and K. Lipnikov. An adaptive algorithm for quasioptimal mesh generation. *Computational Mathematics and Mathematical Physics*, 39(9):1468–1486, 1999.