An Implementation and Evaluation of a Co-rotational Finite Element Method on Mobile Architectures

by

Giovanni Campo, B.Eng.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

(Interactive Entertainment Technologies)

University of Dublin, Trinity College

September 2015

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Giovanni Campo

August 30, 2015

Permission to Lend and/or Copy

Acknowledgments

I would like to express my sincere gratitude to my supervisor Dr. Michael Manzke for taking me on this dissertation and for providing me with all the necessary facilities for the research.

I also wish to express my deepest appreciation and gratitude to my friends, Antonio Nikolov, Sarah Noonan, Patrick O'Halloran and Tony Cullen, for their invaluable help throughout the Masters.

Words cannot express my appreciation and love for my Mom for giving me constant support and encouragement.

Last but not least I am extremely thankful to my friend Darren Caulfield, for his constant support and help, for sharing with me his immense knowledge and for generously proofreading this dissertation.

Giovanni Campo

University of Dublin, Trinity College
September 2015

An Implementation and Evaluation of a Co-rotational Finite Element Method on Mobile Architectures

Giovanni Campo
University of Dublin, Trinity College, 2015

Supervisor: Dr. Michael Manzke

Finite element methods (FEM) have been an active area of research for physical simulations over the last 30 years. FEM is mainly used to simulate deformation and fractures of solid objects. Its application is of particular interest in engineering and scientific fields, where accuracy is more important than plausibility. However, due to its complexity, it is only suitable for offline simulations. Notwithstanding these limitations, FEM can be used for interactive applications. Earlier work has shown the feasibility to run FEM in real-time contexts, on limited console hardware, using linear tensors. Instability problems, which arose from the use of linear approximations, were successfully addressed using a co-rotational formulation.

This dissertation explores the viability of achieving a robust and real-time FEM implementation on mobile architectures. A co-rotational FEM is fully implemented on both CPU and GPU hardware. Experiments are conducted to benchmark and to evaluate the efficiency of memory hierarchy on the Tegra architecture. The results are promising, showing interactive frame rates on both CPU and GPU implementations.

.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Real-world phenomena are very complex to model, both mathematically and physically. Ordinary or partial differential equations are used to describe these physical events. Solving a differential equation can be an expensive task, especially for partial differential equations. In computer graphics it is possible to run simulations either using a *physical simulation* approach or a *physically based animation* approach. Typically, the latter is only a close approximation to real physics, where the goals are plausibility and visual appeal. This method is more suitable for real-time, e.g., interactive, environments. In contrast, physical simulations aim for accuracy, with the intent of recreating a physical system that is as close as possible to the real physics model, and are of particular interest in engineering and scientific fields. In recent years, there has been a breakthrough in research that has made possible the use of algorithms from physical simulations, specifically finite element methods (FEM), in real-time contexts. This goal was achieved using a simpler analytical representation of the real-world phenomena to model, e.g., by adopting linear approximations and solving partial differential equations of the first order. Previous work [4, 5, 6] addressed the instability problems that arose with the adoption of linear Cauchy strain tensors, which were successfully addressed by using a co-rotational formulation.

Parker *et al.* [7] were the first to develop a system to model deformation and fracture of rigid bodies in real time using the co-rotational FEM. The system was successfully used in the Xbox 360, Playstation 3 and PC versions of the video game *Star Wars: The Force Unleashed* published by LucasArts. Their algorithm was robust and fast enough to provide a real time simulation of deformable materials that the user can interact with.

Their work was remarkable: the complexity of a whole system comprising a FEM formulation, time integration, fracture, collision detection and response was efficiently implemented on limited console hardware from 2005.

Nonetheless, the Xbox 360 architecture was unique in its genre at the time. It employed the first chip multiprocessor (CMP) and used more than 2 cores (3 in fact), being the first console with a unified shader architecture [8]. Although it was a revolutionary console in 2005, today's chips are significantly faster: CPUs contain more than 4 cores with a more efficient MIPS-per-watt ratio. As an example, the new *Tegra X1* NVIDIA mobile processor released in 2015 is based on Maxwell GPU architecture and has 256 streaming processors (192 more than the Xbox 360). Despite it being a mobile processor, it outperforms previous generation console architectures (Playstation 3 and Xbox 360) in terms of computational power and energy efficiency. Its predecessor, the chip *Tegra K1*, is based on the Kepler GPU architecture. Although less powerful than the Tegra X1, it still outperforms Playstation 3 and Xbox 360 with its 192 streaming processors (SPs). Those mobile architectures are extremely powerful and flexible. They are currently used by NVIDIA for deep learning, automotive visual computing and computer vision.

Recent work by Muller *et al.* [9] proposed an alternative approach that models destructions using a novel method based on a volumetric approximate convex decompositions. Although this approach avoids the cost of stress analysis and crack propagation, and is fast enough to be used in gaming contexts, it cannot model deformation of solid objects.

Very little previous work has been done in the area of evaluating and implementing complex algorithms using mobile architectures. Nikolov [10] investigated the performance of ray-casting volume rendering on mobile devices using OpenGL ES shaders and CUDA. Results showed that the CUDA implementation performed significantly slower than its shader equivalent. Nonetheless, real time performance was achieved.

Applications of FEM, although not fully exploited in games, are found in many scientific fields. With the advent of high-performance processors on mobile and embedded platforms it is worth analysing whether or not it is possible to run complex algorithms (e.g. soft body simulation, volume rendering, large parallel tasks, etc.) in real time.

1.2 Objectives

The main aim of this dissertation is to benchmark and to evaluate the performance of a co-rotational FEM algorithm for isotropic materials on mobile architectures. The NVIDIA *Shield Tegra*, and the Jetson TK1 based on the Kepler architecture, are the first mobile and embedded solutions to employ a full GPU architecture. We therefore use to evaluate the experiments.

It is also of interest to perform an analysis of the memory hierarchy, the influence of the memory access pattern and the impact of the register pressure, which are of great importance for achieving a better understanding of how a mobile GPU architecture reacts under stress.

Ultimately, the goal is to explore the viability of running soft-body simulations of medium-complexity scenes at interactive frame-rates.

Chapter 2

State of the Art

2.1 Soft-Body Simulation

In general, to simulate motion and physics dynamics there exist two central methods: rigid-body and soft-body simulation.

Soft-body simulation has been studied for more than three decades [11]. Deformable models simulate non-solid objects such as cloth, hair, elastics and liquids in computer graphics, and are used for special effects in movies and video games. In soft-body simulations the relative distance of two points in a deformable model is variable. Vertex rearrangement during collisions with the environment makes soft-body simulation more complex than rigid body-simulation [12].

In real time environments, soft-body simulation, due to its complexity, is an approximation of the real physics model, providing only visually plausible results. The choice is to adopt models that appear plausible instead of being strictly accurate. The idea is to trick the observer, who can be easily fooled if the simulation looks convincing. The goal is to achieve at the right experience and not the correct physics. While plausibility is acceptable in video games or special effects, it is usually not sufficient for accurate scientific/engineering simulations like structural analysis, surgery simulations, etc. In contrast to computational sciences where the main focus is on accuracy, the main issues are stability, robustness and speed, while the results should still remain visually plausible [13].

2.2 Mass-Spring Systems

There exist a variety of techniques to run simulations of soft bodies. Mass-spring systems, for example, have been used widely to efficiently model deformable objects in real time. A volume mesh consisting of a collection of point masses connected by springs in a lattice structure is produced to represent an object [11].

Objects of soft bodies using mass-spring systems can be discretised into volume meshes using a 2D grid structure, in the case of cloth simulation, for example, or a 3D structure. The volume mesh can be triangular, rectangular, or a tetrahedral representation where each point has its own properties such as mass, velocity, force and position [11].

Mass-spring systems are computationally less expensive compared to *continuous models* and *finite element analysis* and are suitable for processing large amount of data or large volumes of objects. In particular, they can be used in cloth [14] and hair animation, facial animation [15] and interactive surgery applications [16].

These systems are not a panacea and thus have some drawbacks. The discrete model is only an approximation of the physics that occurs in a continuous body [11]. The spring connections are usually not derived from material properties and proper values for these constants are not accurate. Moreover, a mass-spring system with explicit integration, e.g. Euler or Runge-Kutta, cannot handle increased stiffness[1] under large spring constants. Stiffness causes poor stability and requires the adoption of small time steps in the integration method.

Although explicit integration methods are faster than implicit methods, when they are applied to stiff systems they show instability and thus are not robust [17]. Therefore, using explicit solvers results in an unreliable simulation.

There exist different approaches to tackling the problems discussed so far. One is using implicit solvers, e.g. *Verlet integration.* Verlet integration is a numerical method

[1]An ODE is stiff when certain numerical methods for solving the equation are numerically unstable, unless the step size is taken to be extremely small.

to integrate Newton's equation of motion, used since the 1960s to model molecular dynamics. Verlet is numerically stable and computationally cheap to calculate and is different to explicit integration. It uses a velocity-less representation and a different integration scheme. Instead of storing position and velocity, it stores current position x and its previous position x^*.

This methodology was firstly introduced in physical-based modeling by a mathematician and programmer named *Thomas Jakobsen*. His work was used for the first time in the IO Interactive's game *Hitman: Codename 47*, published by Eidos Interactive in the year 2000.

2.3 Position-Based Dynamics

A more reliable and robust method to run simulation of dynamic systems is by using position-based dynamic frameworks. They base their concept on implicit integration and are more stable than mass-spring systems. They allow for the easily resolution of collision constraints while resolving penetration violations completely by projecting intersecting points to valid locations. They use the current and previous position of particles or vertices of a mesh, e.g., a Verlet-based integrator, which bypasses the force and velocity layers and directly modifies the positions. The mass-spring model is then converted into a system of constraints and considered a mass-less particle system [18].

A non-linear Gauss-Siedel solver is used to compute constraints one by one. In spite of the approach being simple and unconditionally stable, and providing a high level of control over the simulation process, it suffers from the drawback of propagating the information slowly through a mesh. The slow convergence lets soft bodies look "stretchy" producing undesirable effects, i.e., visual artefacts [18].

Convergence can be increased by using global solvers. The Newton-Raphson solver is a method to solve non-linear systems of equations but it is both computationally very expensive and complex to code. Therefore, it is not practical for real time applications [18].

Position-based methods are not as accurate as force-based methods in general, where positions evolve through numerical integration of accelerations and velocities. However, they provide visual plausibility. Therefore, applications of these approaches still remain popular in virtual reality, computer games and special effects [19].

2.4 Force-Based vs Geometry-Based Methods

To summarise the two techniques discussed so far, for deformable object simulation-mass-spring systems and position-based dynamic frameworks-it can be said that *force-based methods* are based on Newton's second law of motion, and they remain the most popular approach in computer graphics. They are computationally cheap to compute but they suffer from stability problems [19]. It is very difficult to tune the spring constants, and in combination with a poorly designed spring network set-up it can be hard to achieve the desired behaviour of the object. Moreover, mass-spring networks cannot capture volumetric effects directly, such as volume conservation or prevention of volume inversions [19].

In contrast, *geometry-based methods* omit the velocity layer operating on the positions. The main advantages of a position-based approach are its controllability, unconditional stability, robustness and speed [19]. These methods are mainly used in interactive applications, where the aforementioned properties are more important than accuracy, e.g., for cloth and fluid simulation, as well as soft-body dynamics. There exist a variety of alternatives in the literature that go beyond plausible simulations, allowing physically accurate simulations. Applications of position-based methods are interactive surgical simulation, where Wang *et al.* [20] introduced a mass-spring model based on a surface mesh to simulate deformable bodies in real time. The surface model is coupled with a rigid core by using spring forces. This rigid core is simulated using shape matching, which results in a fast and stable simulation [19, 20].

Rungjiratananon *et al.* [21] proposed an approach based on *Lattice Shape Matching*, originally introduced by Rivers and James [22], to simulate complex hairstyles using a shape-matching approach. A chain of particles, subdivided in overlapping chain regions, represents each hair strand. A position-based strain is applied to each strand after shape matching, which moves the particles in the direction of their root. To realise different hairstyles, initial configurations are used and region sizes of a chain are modified.

A wide range of physics frameworks including *PhysX*, *Havok Cloth*, *Maya nCloth* and *Bullet*, implement position-based dynamics. While predominantly used in real-time applications, position-based dynamic is also often used in offline simulation. However, the desirable qualities of PBD comes at the cost of limited accuracy because the method is not rigorously derived from continuum mechanical principles [23].

2.5 Continuum Mechanics

There exist more-accurate physical models to deal with soft-body simulation. Viewing the system as a *continuum* is one of them. Unlike the discrete mass-spring models, continuum models are derived from equations of continuum mechanics.

The continuum model of a deformable object considers the equilibrium of a general body acted on by external forces. The object deformation is a function of these forces and the object's material properties. The object reaches equilibrium when its potential energy is at a minimum [11].

$$\Pi = U - W \tag{2.1}$$

Equation 2.1 denotes the total potential energy of a deformable system, where U is the total strain energy of the deformable object, and W is the sum of the external forces acting on it. When forces are applied on a soft body, an energy, i.e, the strain energy is stored in the body as material deformation. The strain energy is the energy stored in the body as material deformation. The load acting on the system, e.g., the external forces are due to three sources [11]:

- Concentrated loads applied at discrete points
- Forces acting on the body, as gravitational forces
- Forces distributed over the surface of the object, such as pressure forces

To find a solution to the partial differential equation 2.1, a numerical method is needed, such as *Finite Element Methods* (FEM), *Finite Differences Methods* (FDM), or *Finite Volume Methods* (FVM).

2.6 Finite Element Analysis

Finite element analysis (FEA) originated from the need to solve complex elasticity and structural problems in civil and aeronautical engineering. The mesh discretisation of a continuous model into a set of discrete sub-domains, called *elements* and connected at discrete points called *nodes*, is the principal characteristic of FEA.

FEA consists of three main steps:

1. *Preprocessing*: The geometry of a model is tessellated into a number of elements connected to nodes. There exist two types for these elements: tetrahedra or hexahedra.

2. *Analysis*: The elements produced in the previous step are used as input to the finite element implementation, which constructs and solves a system of linear or nonlinear equations.

   $$K_{ij}u_j = f_i \tag{2.2}$$

   u and **f** are respectively the displacements and externally applied forces at the nodal points while **K** is the stiffness matrix.

3. *Postprocessing*: Displacements of the simulation mesh are mapped back to the original geometry to reflect topological changes.

2.6.1 Finite Differences Method

The finite-differences methods (FDM) are numerical methods for approximating the solutions to differential equations using finite-difference equations to approximate derivatives. Partial differential equations (PDEs) differ from ordinary differential equations (ODEs) in that they have two or more independent variables.

There exist several numerical approaches to approximate solutions of PDEs. PDEs are approximated by discretising the spatial dimension and they are successively converted into ODEs. ODEs can either be solved by other methods such as Runge-Kutta or solved by Euler integration methods.

Specifically, the spacial dimension of the material is divided into a regular lattice and then numerical differencing is used to approximate the spatial derivatives required to compute the strain and strain tensors. However, irregular structures makes his approach too complicated and thus it suited mainly for problems with a regular structure [24].

The same approach taken to derive a numerical approximation for ordinary differential equations can be applied to partial differential equations.

2.6.2 Finite Element Method

A finite element method (FEM) is characterized by a variational formulation, which is a discretisation strategy, one or more solution algorithms and post-processing procedures. The discretisation strategy subdivides the continuum of the material into distinct elements as shown in figure 2.1. Within each element, a local function, which is also called a *shape function*, describes the material. These shape functions associate the elements with the vertices they contain within. Adjacent elements will have nodes in common, so that the mesh defines a piecewise function over the entire material domain [24].

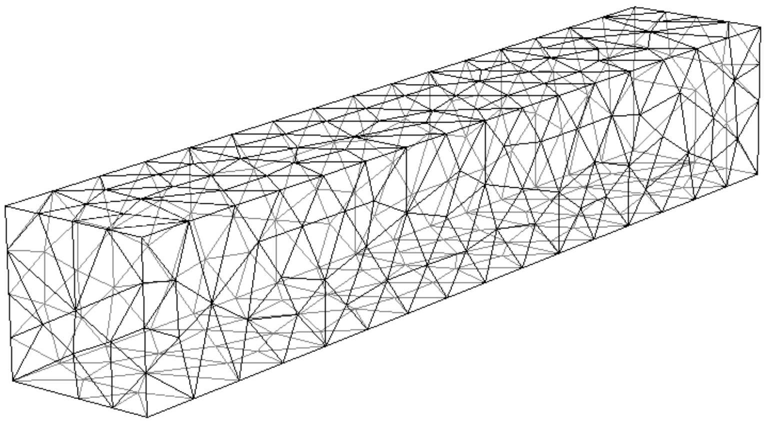


Figure 2.1: Tetrahedralized rectangle. The internal structure is shown.

Examples of variational formulation are the *Galerkin method*, the *discontinuous Galerkin method*, *mixed methods*, and so on. Discretisation transforms the partial differential equation of motion into a system of ordinary differential equations which are easier, to some extent, to solve.

FEM has been an active area of graphic research over the last decade, for both real time and offline applications and specifically in the context of fracture and deformation. Key concepts in the area of physically-based dynamics of flexible materials were originally introduced by Terzopoulous *et al.* [25] using finite difference approximation, while in the area of fractures O'Brien *et al.* [24, 26] simulate ductile and brittle fracture propagation based on a stress map derived using the finite element method.

FEM is physically more accurate than mass-spring models or position-based dynamics. The object's deformation behavior can be specified using material properties, which can be looked up in textbooks instead of tweaking spring constants, and the force coupling between mass elements is defined throughout the volume rather than according to the spring network.

Nevertheless FEM is computationally more expensive. However, FEM methods have been used in real-time applications by discretising 3D meshes, typically with polyhedral elements, assuming isotropic surfaces and linear elasticity.

While not suitable for engineering analysis, such models are sufficient to obtain visually plausible results [5]. Such first-order approximation for FEM is cheap and thus feasible in real time due to the lower cost of a linear strain measure. However, it is only applicable for modelling small deformations accurately.

One important feature of the linear approach is that the stiffness matrix of the system is constant and numerically well conditioned, yielding fast and stable simulations. However, the linear model is not rotationally invariant. The linear model is only a first-order approximation at the undeformed state and therefore under large deformation objects increase unnaturally in volume [5]. To solve this problem accurately, non-linear elasticity could be taken into account. However, the stiffness matrix is no longer constant; therefore it needs to be recalculated at every step as the Jacobian of the non-linear function that describes the internal elastic forces. Such a matrix is non-trivial to calculate as its dimension can reach 81 elastic constants that are independent of stress or strain [27]. Thus, in real-time systems large deformations are usually avoided in favour of small displacements.

Muller *et al.* [5] describe a method for decomposing deformation into separate rigid-body and strain components. By warping the constant stiffness matrix of the system along with a rotational field it is possible to use a linear approach, thus permitting the use of fast solutions in context of large deformations. However their node-centric decomposition produced undesirable ghost forces [7].

Tetrahedral Geometry

An often-used tesselation for volumes is a tetrahedral mesh. This type of discretisation approach is very common in the graphics literature, and can be found in several papers: [5, 24, 26, 7]. The elements in the FEM are represented by tetrahedra; thus when a mesh is generated, the domain of the original mesh is sub-divided into tetrahedral finite elements. Tetrahedra are in general a good approach to approximate arbitrary volumes, especially when this formulation is used to generate fractures. When tetrahedra are split along a fracture plane, the resulting pieces can be decomposed exactly into more tetrahedra [24].

The process of partitioning the space into regions, in the 2D case, is called a Voronoi diagram. The main condition is that in each region R_k, the distance (which can be either the Euclidean or Manhattan distance) of each point X, contained in R_k, to the Voronoi center P_k is less than the distance to any other Voronoi vertex P_j for j different from k (see figure 2.2. There exist several algorithms to generate a Voronoi diagram:

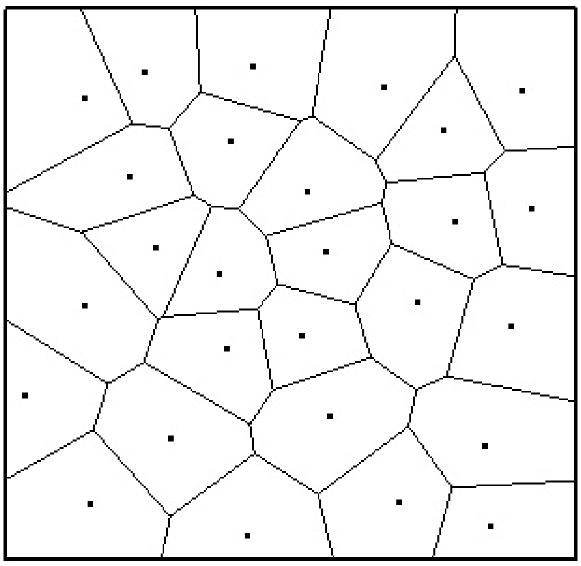


Figure 2.2: Voronoi diagram.

- Divide-and-conquer $O(n^3)$ worse-case
- Sweep Plane $O(n^3)$ worst-case
- Incremental Insertion $O(n^2)$ worst-case

All of the aforementioned algorithms have a $O(n^2)$ complexity which can be quite slow in some scenarios when the geometry being processed is particularly complex. Fortune [28] presented in his paper a novel method based on a sweepline technique that runs in $O(nlog(n))$.

A 2D Voronoi diagram can be extended to three dimensions. Higher-order diagrams can be generated recursively based on the $(i-1)th$ order. However 3D Voronoi diagrams are not suitable in case of concave or irregular shapes. The main reason is the reliability of the visual appearance of filling the holes once the mesh has been split. There is another technique, the Delaunay triangulation, which triangulates the space based on the conditions that a set of points P in a plane must lie on the perimeter of the circumcircle of any triangle i.e., they must not be contained in any triangle. (See figure 2.3)

The Delaunay triangulation of a point set P corresponds to the dual graph of the Voronoi diagram for P. Vertices of a Delaunay triangulation are the centers of each

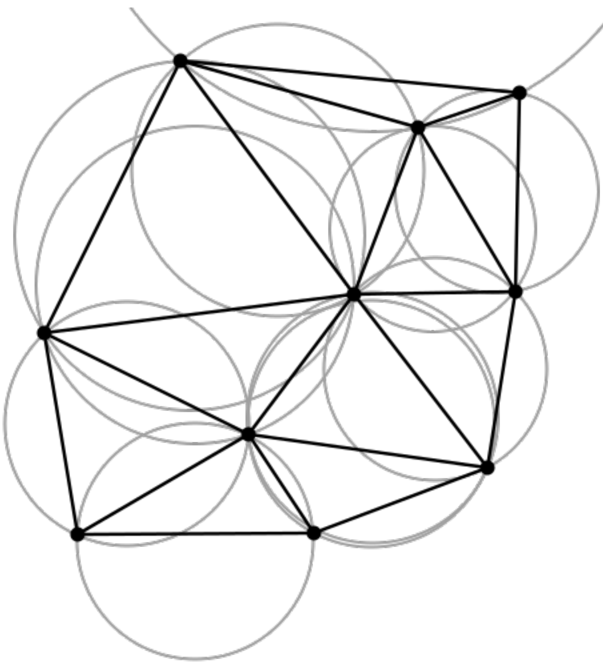


Figure 2.3: Delaunay triangulation

cell in a Voronoi diagram.

Delaunay triangulation works quite well in 2D space, although for usage in real-time applications a 3D implementation is required. Tetrahedra can be generated using the high-order Delaunay tetrahedralisation. The Delaunay tetrahedralisation is the 3D equivalent of triangulation. Instead of generating triangles, it generates tetrahedra. A given polytope is decomposed into non-overlapping tetrahedra, where the vertices of the tetrahedra must be vertices of the original polytope. The condition that must be satisfied is that the four points of each tetrahedron must lie on the perimeter of a circumsphere.

Often, give the mesh geometry, the domain boundaries are not respected, e.g., inconsistencies are caused by co-planar vertices so the Delaunay condition is not satisfied. These inconsistencies are removed by facet re-meshing and Steiner point insertions (e.g. introducing additional vertices). There is not a large amount of literature about Delaunay tetrahedralization (DT). The problem in 3D is far from being solved. There are several algorithms that solve the problem but only one that has square complexity in the worst-case scenario: *Incremental insertion* [29]. With this algorithm, each point is inserted one at a time in a valid Delaunay triangulation and the tetrahedralisation is updated, with respect to the Delaunay criterion, between each insertion [29]. However, the discretization step is usually performed offline to save computational time.

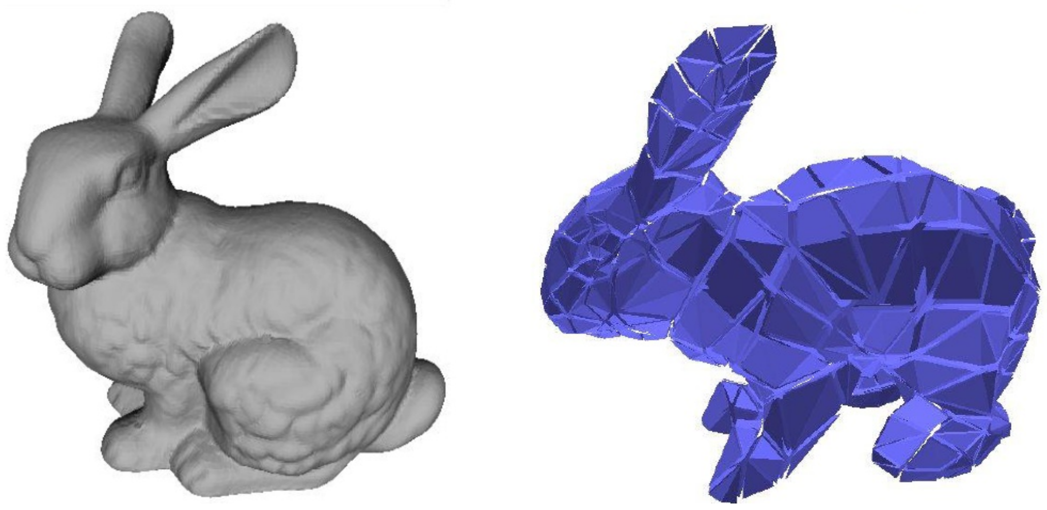


Figure 2.4: A Delaunay tetrahedralization example.

A tetrahedra is defined by four nodes, labelled 1, 2, 3 and 4, with reference position $\vec{X}_1$, $\vec{X}_2$, $\vec{X}_3$, $\vec{X}_4$ (see figure 2.5), a position in the world coordinates, p, and a velocity in world coordinates, v.

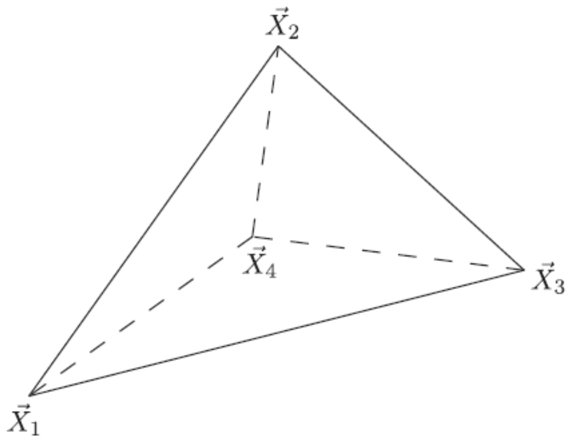


Figure 2.5: A tetrahedron.

Among volumetric geometry representations, tetrahedral meshes are the most used. The simple interpolation methods that tetrahedral discretisation imply, make them the most convenient to use. Thus, the reconstructed deformation map $\hat{\phi}$ can be defined to be a piecewise linear function over each tetrahedron. In each tetrahedron $\hat{\phi}$ can be defined as follows [30]:

$$\hat{\phi}(\vec{X}) = A_i\vec{X} + \vec{b_i} \ \forall \vec{X} \, \epsilon \, T_i$$

The interpolation scheme implied by the equation above is a barycentric interpolation on every element. The differentiation, with respect to $\vec{X}$, reveals that the deformation gradient $F = \partial\hat{\phi}/\partial\vec{X} = A_i$ is constant on each element and as a consequence any discrete strain measure and stress tensor is constant too. Linear tetrahedral elements are also referred to as constant strain tetrahedra [30].

Co-rotational FEM

There are several methods to achieve real-time performance using FEM. A very common approach is to adopt a linear stress tensor, or Cauchy's tensor. Although a linear approximation is cheaper to compute it is not rotationally invariant. When elements undergo large deformations or rotations, artefacts are produced which make the element artificially inflate leading to unrealistic results. A non-linear Green's strain tensor, which is rotationally invariant, would solve this problem but it would lead to a non-linear algebraic system needing to be solved, making the FEM unsuitable for real-time contexts.

Muller *et al.* [5, 4] were the first to introduce the FEM to computer graphics and to address these instabilities. They formulated a geometric approach to separate, on a per-vertex basis in [5] and on a per-element basis in [4], rigid-body and strain components. In general, in the finite element literature, the basic notion of separating out rotation from the rigid body motion is called a *co-rotational formulation*:

A co-rotational method factors out the rotation on a per-element (i.e., per-tetrahedral node) basis. Two configurations are considered: the base configuration u_0 and the co-rotated configuration u_r. The former is the origin of displacements and will not change during the simulation. The latter varies from element to element and is obtained

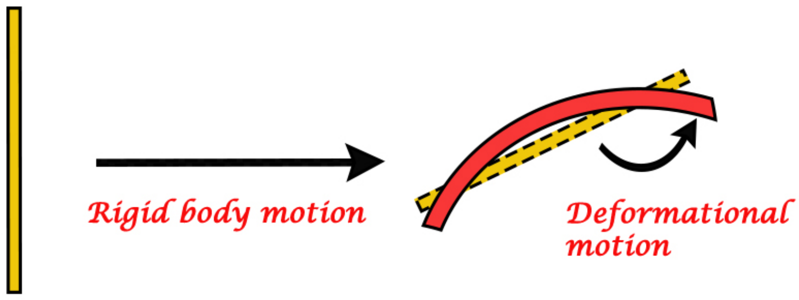


Figure 2.6: Rigid-body motion separated from the deformational motion

through a rigid-body motion of the element from the base configuration. Element displacements $u_r - u_0$ are calculated with respect to the co-rotated configuration [31].

The co-rotational formulation is a good trade-off between the Cauchy and the full non-linear Green's strain tensor. It is a linear approximation and it accounts for the geometric non-linearity by respecting per-element rotations in the strain computation [32]. There are several approaches to extracting the rigid-body motions of tetrahedral elements.

Other approaches are the polar and the QR decomposition, which operates on a per-element basis. QR decomposition has been developed by Nesme *et al.* [6] but introduces vertex-ordering dependent-anisotropies [32]. With the adoption of a QR decomposition in the co-rotational FEM it is possible to achieve a quasi-stable simulation. The QR decomposition is based on the Gram-Schmidt orthogonalization algorithm[2].

The FEM formulation implemented for this work uses the geometric per-element approach described in [4].

[2]It is the decomposition of a matrix A into a product $A = QR$.

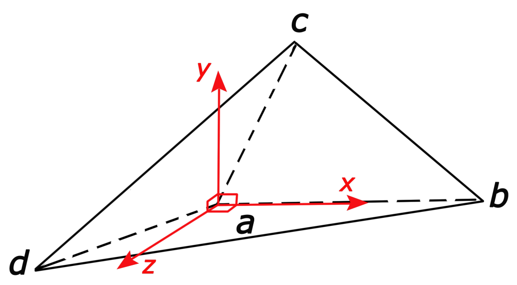


Figure 2.7: Example of QR decomposition. x is chosen to be on the first edge ab while z is orthogonal to the first plane composed by the vectors ab and ac. The last axis y is obtained by construction of an orthonormal frame.

Chapter 3

Experiments

3.1 Overview

In all of the experiments carried out in this work the raptor model represented in figure 4.2 has been used to simulate deformations. As already mentioned in previous chapters, two meshes are used to run FEM: the surface mesh, i.e. the visual representation of the model, and the simulation mesh. The surface mesh has approximately 78k triangles, while the simulation mesh is coarser with only 8431 tetrahedra elements. Each profiling session was recorded over 800 frames.

In the scene there is also a plane represented by a flat surface with a simple texture. Gravity is applied as a base force with an acceleration of $9.8m/s^2$. In order to stress the hardware and cache, every 200 frames an external force of random intensity is applied to a random particle on the simulation mesh. Self-collisions are not taken into account. However, the simulation mesh can collide with the plane where the main model lies. Some fixed particles are applied to the feet of the raptor, meaning that the corresponding tetrahedra will not be affected by the physical deformation. This was done to keep the model in a specific point at the scene.

To evaluate the performance, three core FEM functions were monitored:

- Displacement mapping: this is the barycentric interpolation of tetrahedra coordinates to surface mesh vertices. This step is necessary to map the tetrahedra deformation to the vertices of the surface mesh.

- Compute and add forces: in this step all the internal forces and the product of their stiffness with a position change are computed.

- Conjugate gradient solver: it is an iterative method to find a solution of systems of linear equation. This method solves the differential equation that relates stiffness matrix, forces, position change and deformations of the solid object. An acceptable value for the maximum number of iterations in the conjugate gradient is a number between 30 and 100 [27]. However, it is possible to achieve plausible simulations with lower values, e.g., between 10 and 25. The conjugate gradient is explained in detail in chapter 4.1.6 at page 37.

A set of methods was implemented and adapted to perform an implicit and explicit finite element solver on two different platforms: Windows and Android. The reference implementation is based on the work of Allard *et al.* [27]. It uses a Cauchy linear tensor and adopts a co-rotational formulation. Experiments are run using CUDA on the GPU and a full serial implementation on the CPU.

3.2 Target Architectures

The FEM has been implemented for both CPU and GPU using CUDA. Experiments were conducted running simulations on four different architectures. (See table 3.2 for details).

3.2.1 CPU Architecture

The NVIDIA Shield employs an ARM Cortex A15, a *Reduced Instruction Set Computer* (RISC) architecture, which is comprised of four cores with a maximum clock speed of 2.5GHz, 32KB of L1 instruction and data cache per core for a total of 64KB L1 cache. See figure 3.1. The L2 cache is 16-way set-associative of configurable size with the *Snoop Control Unit* SCU. The SCU is clocked synchronously and at the same frequency as the processors. It maintains coherency between the individual data caches in the processor. The SCU contains buffers that can handle direct cache-to-cache transfers between processors without having to read or write any data to the external memory system [33].

Architecture	**Parameter**	**Desktop**	**Mobile**
CPU	**Name**	**Intel i7 4930k**	**Tegra ARMv7 Cortex-A15**
	Clock Speed	3.4 GHz	2.5 GHz
	# of Cores	6	4
	# of Threads	12	4
	L1 Cache Size	64 KB	64 KB
	L2 Cache Size	2 MB	512 KB - 4 MB
	L3 Cache Size	12 MB	N.A.
	Instruction Set	64 bit	64 bit
	Max Power Consumption	130W	0.35W
	RAM	16 GB	2 GB
GPU	**Name**	**Kepler GK110A**	**Kepler GK20A**
	CUDA Cores	2880	192
	Clock Speed	1.25 GHz	0.85 GHz
	Memory Size	3 GB	2 GB Shared Memory
	Max Power Consumption	250W	2W
	CUDA	6.0	6.0
	OpenGL	4.4	4.4

Table 3.1: CPU and GPU architecture comparison

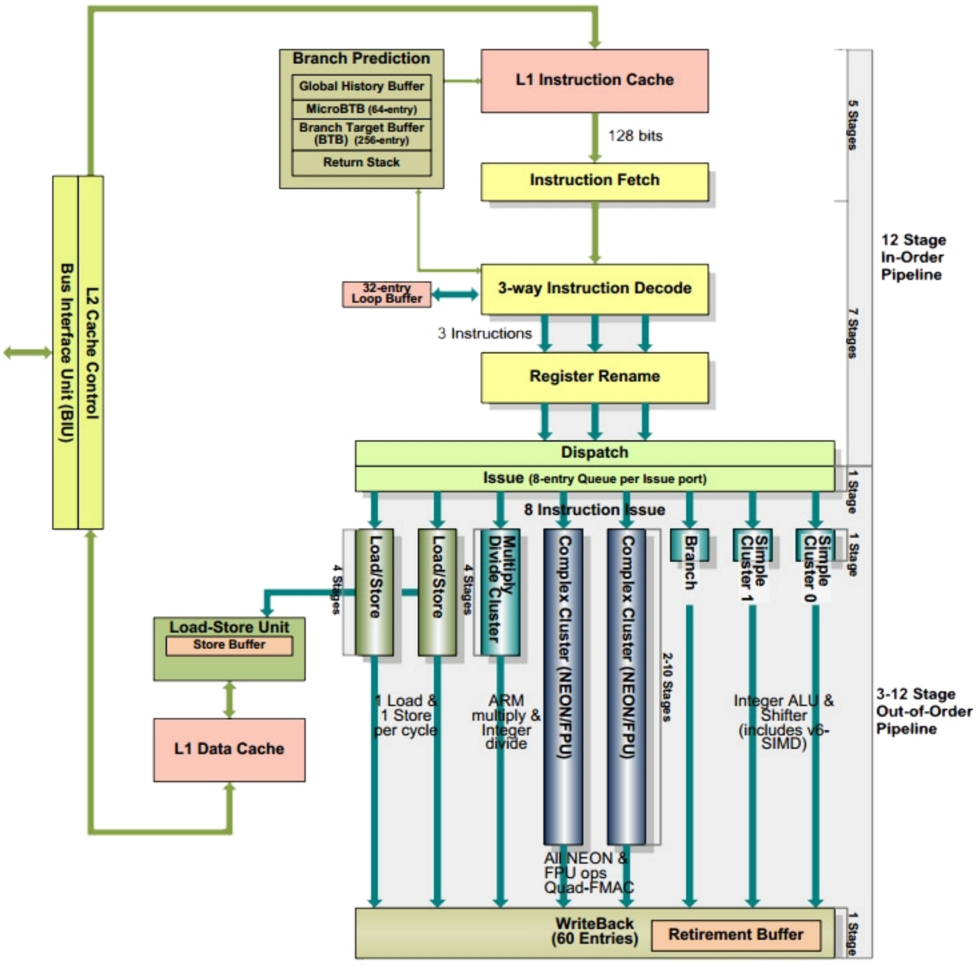


Figure 3.1: Cortex A15 diagram block

The ARM A15 supports a *Single Instruction Multiple Data* (SIMD) instruction set, which is referred to as *NEON engine*, for integer and single-precision floating-point vector operations on double word and quad word [33].

3.2.2 Kepler GK110 Architecture

The CUDA FEM implementation has been evaluated using the *GeForce 780ti* as the desktop reference. Its GPU is based on the Kepler GK110 architecture. It is comprised of 7.1 billion transistors, for a total of 2880 streaming processors (or *CUDA cores* in NVIDIA jargon) laid out in 15 streaming multiprocessors (SMX)- see figure 3.2. It has a power consumption of approximately 17W per SMX for a total consumption of 250W.

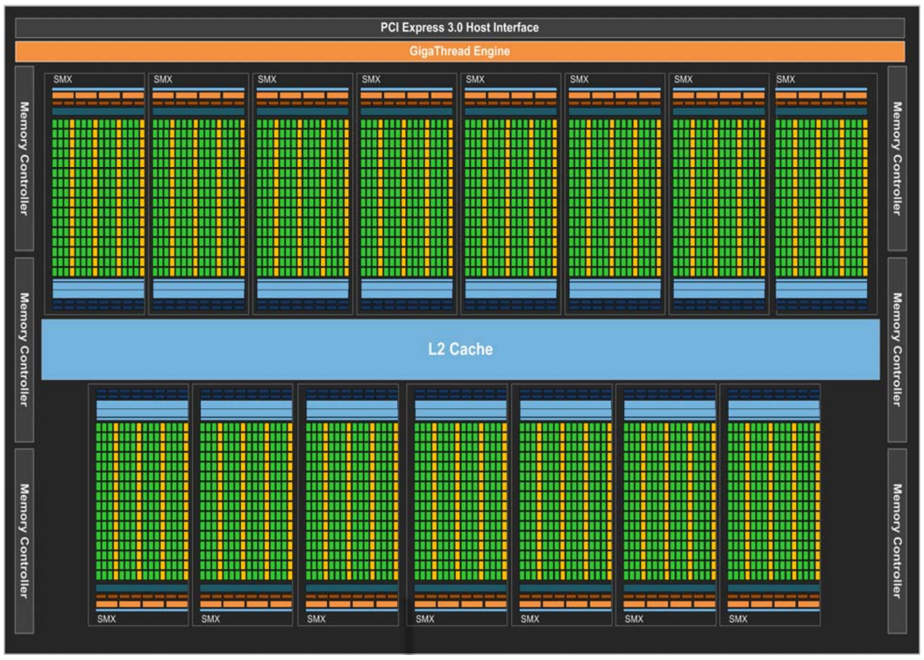


Figure 3.2: Kepler GK110 Full chip block diagram from [1]

The mobile Kepler Tegra GK20A architecture is very similar to the Kepler GK110, which is why we have chosen it as the desktop reference version. Tegra employs the same memory hierarchy of GK110 with two exceptions: the absence of 48KB read-only cache for each SMX accessible by the *Texture Unit* (see figure 3.3 and figure 3.5) and the absence of 64 double precision floating-point units.

Double precision floating-point calculations are only used in scientific computing applications like fluid dynamics, molecular dynamics, model fitting and data analysis. 64-bit data uses twice as many bits as single precision; therefore, it uses twice as much RAM, cache, and bandwidth, thereby reducing the overall system performance. However, single precision only is used to run the tests. In our tests we use only single-precision operations.

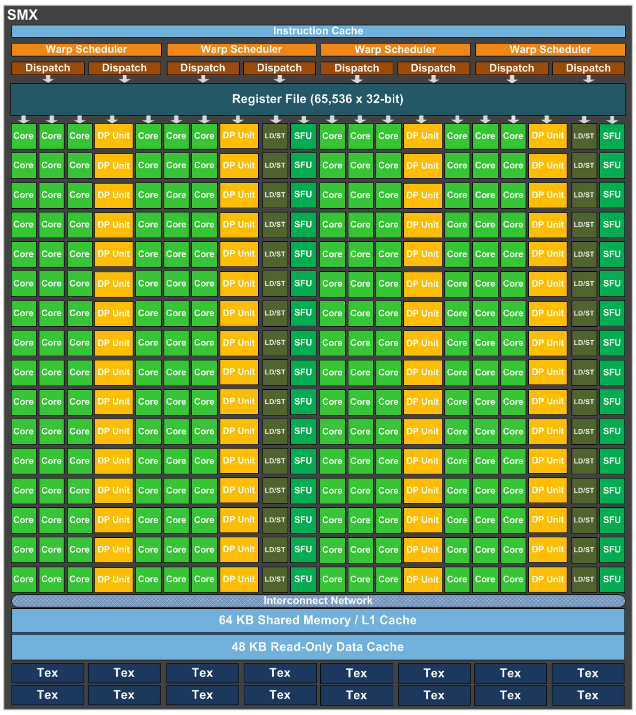

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Figure 3.3: SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST) from [1]

Warp scheduler

The SMX schedules threads in groups of 32 parallel threads called *warps*. Each SMX features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently (see figure 3.4). Kepler's quad warp scheduler selects four warps, and two independent instructions per warp can be dispatched for cycle [1].

3.2.3 Kepler GK20A Architecture

The Kepler GK20A architecture utilizes 192 CUDA cores laid out in a single SMX. The main differences with the desktop version were discussed in section 3.2.2.

The GK20A GPU is organized in Graphics Processing Clusters (GPC), SMXs, and memory controllers. It consists of one GPC, one SMX unit and a memory interface. It also includes four Raster Operation Processors (ROPs) and has a 128KB L2 cache between the ROPs and the memory interface (see figure 3.5).

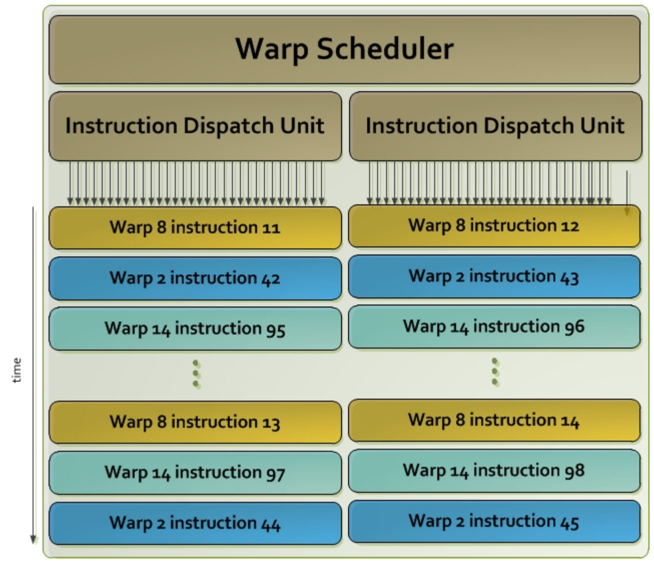


Figure 3.4: Each Kepler SMX contains 4 warp schedulers, each with dual Instruction Dispatch Units [1]

	Xbox 360	**Playstation 3**	**Tegra K1**
GPU Features	DX9	DX9	DX11
GPU Horsepower	240	192	365
CPU Horsepower	3600	1200	5612
Power	100W	100W	5W

Table 3.2: Performance comparison between consoles and Tegra K1 [2]

This architecture, despite being a mobile solution, delivers higher peak shader GFLOPs and higher total CPU throughput than older generation consoles like Xbox 360 and PS3. A performance comparison can be found in table 3.2.3. GPU throughput is based on peak fragment shader GFLOPS of each platform. CPU throughput is instead calculated as the estimated SPECint2000[1] performance multiplied by the number of CPUs [2]. It is interesting to note the power efficiency, which is 95W less (and ~12W per SMX) than the desktop Kepler GPU.

[1]It is part of a standardised set of benchmarks that can be applied to any hardware.

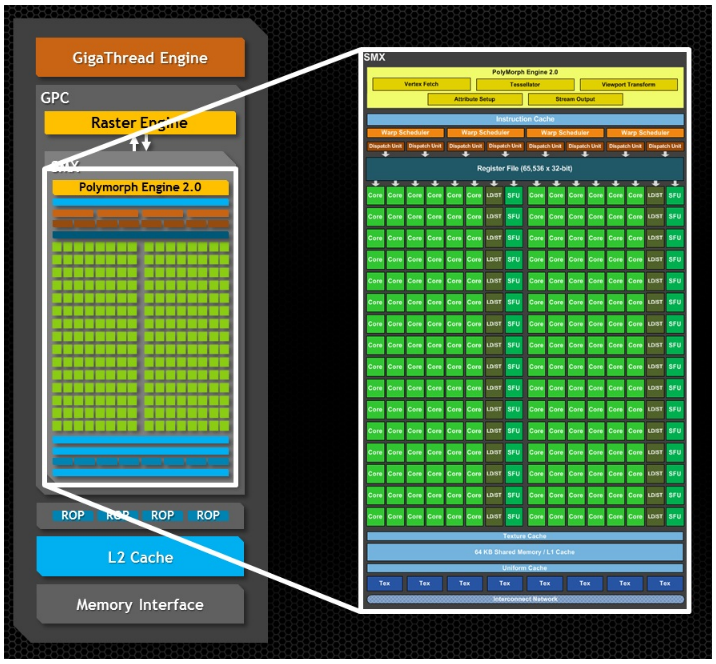


Figure 3.5: Full Kepler GPU (left) and Kepler SMX Unit (right) in Tegra K1 [2]

3.3 CUDA Framework

One of the two FEM implementations uses the CUDA framework on the GPU. CUDA, (Computer Unified Device Architecture), is a parallel computing application programming interface model created by NVIDIA. It enables the execution of general-purpose code on the GPU in tasks where a high degree of parallelism can be exploited. The CUDA platform is designed to work with programming languages such as C, C++ and Fortran. The graphics programming model executes in parallel shader threads independently, while parallel-computing, in order to efficiently compute a result, requires that parallel threads synchronize, communicate, share data, and co-operate [34]. A result data array can be partitioned into blocks and each block into elements, so that the result blocks can be computed independently in parallel, and the elements within each block can be computed co-operatively in parallel.

The CUDA programming language has two main function definitions: global and device functions. The former are launched by the CPU and then they are run by the GPU. Global functions are also known as kernels. The latter are GPU functions and they can be called only from within a kernel, e.g. from the GPU.

Three levels of parallelism are offered: grids, blocks, and threads. A number of threads and blocks can be defined when a kernel is launched. Figure 3.6 shows the different concepts: threads, blocks and grids. Threads have access to a local memory, while threads in a block can access a shared memory to co-operate. A global memory is available and accessible from all threads in different blocks and grids. Specifically, in the GPU hardware threads are executed in parallel by the SPs, while each block is assigned to a SMX. Memories have different speeds. Global memory, which is the main memory of the GPU has two orders of magnitude more latency than on-chip memory. For this reason and because of obvious synchronisation problems it is important to avoid accessing global memory often. Shared memory is much faster than global memory but its bandwidth is considerably lower than that of registers. It can be used as a user-managed cache to reduce the number of slow global memory accesses, communication between several threads within a group so they can collaborate in a given task, or simply to temporally store data and reduce register pressure. Registers are used to store value types, declared within a kernel, or passed as arguments. Warps, which were discussed in the previous section, can run up to 32 threads concurrently.

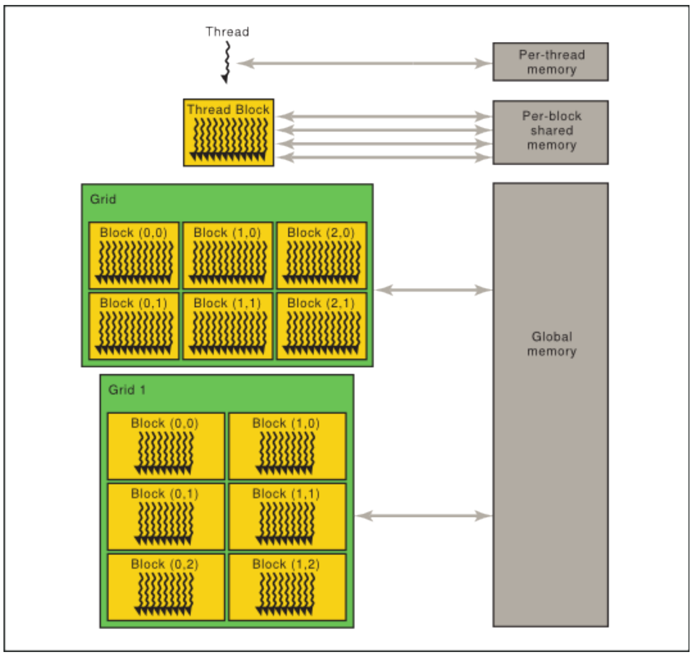


Figure 3.6: CUDA grids, blocks and threads [3]

The main goal in any CUDA implementation is to keep the GPU occupied fully at all times in order to achieve good performance. Other important key factors are: branching, warp occupancy, race conditions and cache reuse. A *Single Instruction Multiple Thread* (SIMT) processor realizes full efficiency and performance when all available threads of a warp take the same execution path [34]. Thus, the use of conditional instructions in a kernel is generally not a good idea. If threads of a warp diverge, the warp serially executes each branch path taken, disabling threads that are not on that path. When the diverged paths complete, the threads re-converge to the original execution path [34]. Having sleeping threads in a warp decreases the warp occupancy, bringing down the overall performance. Race conditions, in multi-threaded programming as well as in GPU parallel programming, occur when a common resource is accessed concurrently by multiple threads. Fortunately, CUDA offers a set of atomic instructions that guarantee locked read/write operations.

In chapter 5, which is about the evaluation of the experiments, all of the aforementioned aspects are taken into account, to assess the performance and the memory hierarchy of the CUDA implementation.

Chapter 4

Implementation

As discussed in chapter 3, a set of methods were implemented for an implicit finite element solver based on the work of Allard *et al.* A significant amount of effort was expended to adapt the algorithm to work on both Windows and Android platforms.

4.1 Pipeline

Figure 4.1 shows the high-level architectural diagram of the designed pipeline that was designed for the FEM. The workflow on the left represents the mesh preparation process. A library called *NetGen* [35] was used to discretise the surface mesh. This process is performed offline as it requires a considerable amount of time to be completed. Details are given in section 4.1.1. Once the tetrahedral mesh has been calculated it is saved into a text file. The workflow on the right represents the main physical simulation. Surface and tetrahedral meshes of a given solid object are read and loaded into memory. The tetrahedral mesh is partitioned using an octree data structure and a displacement map is created to associate vertices to tetrahedra. After the mesh is fully mapped the simulation starts. There are four main functions:

- compute the internal forces that act upon the elemets.
- run the conjugate gradient to solve Newton's equation of motion $M\vec{a}+D\vec{v}+K\vec{u}$ in order to find the tetrahedral displacement $\vec{u}$ for each tetrahedron in the simulation mesh.

- apply an Euler implicit time integration step to update velocities and positions.
- map the new displacements back to the vertices; the formula used is explained in section 4.1.7.

These four steps are explained in detail in the following sections.

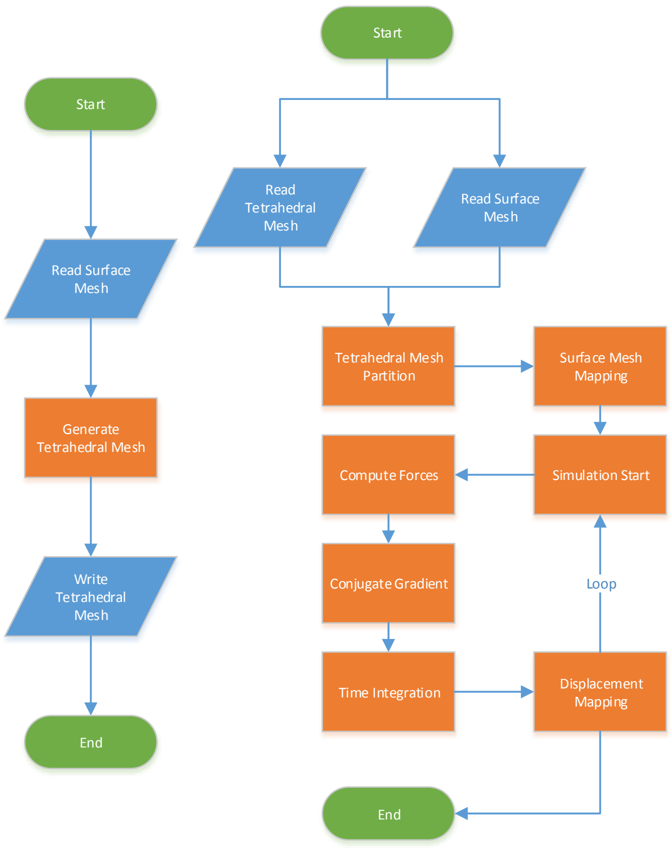


Figure 4.1: Pipeline implementation overview

4.1.1 Mesh Preparation

Mesh preparation is the first step needed before running the simulation. To solve the partial differential equation associated with the dynamic-system description of a deformable body, the domain needs to be tessellated. This process subdivides the surface mesh into a finite set of adjacent and non-overlapping sub-domains, which are either four-node tetrahedra or eight-node hexahedra. The displacement fields of the continuum matter within elements are mapped to their vertices, approximating the associated equations of the object using the displacements of their nodes [36]. Representing a mesh with hexahedra is more accurate. Although objects can be subdivided using fewer elements, hexahedra require greater memory resources (24 DOF versus 12 DOF of a tetrahedra) and thus are computationally more expensive. In this specific implementation the entire surface mesh was fully partitioned using tetrahedra.

With the use of linear tetrahedra, as explained in chapter 2, it is possible to replace the continuous displacement field with the displacement of the tetrahedra. Such displacement maps are called *shape functions* and are used to map a tetrahedron to its vertices. In this specific case, a shape function is based on the linear interpolation of the tetrahedron's barycentric coordinates.

Tetrahedra are obtained by performing a Delaunay tetrahedralisation, which is a non-trivial task; thus, external libraries were used. Mesh mapping is the last step in the pipeline and it is performed in real-time.

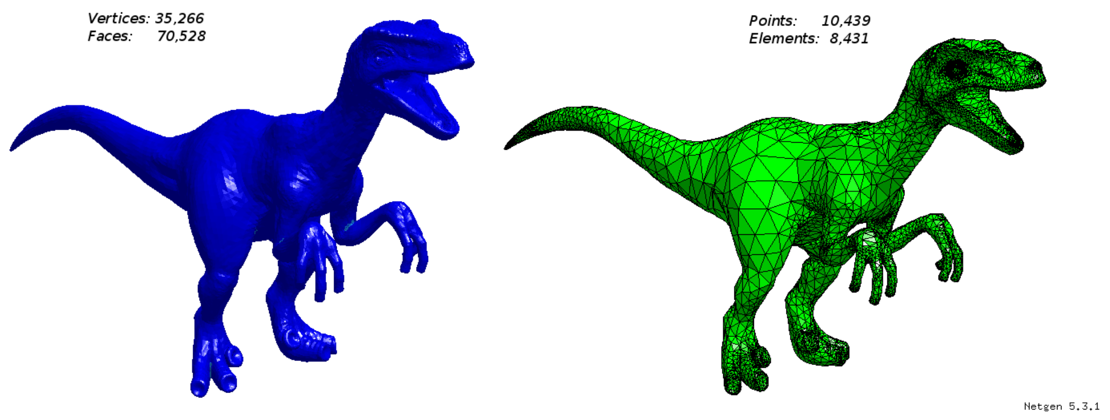


Figure 4.2: Surface mesh left and simulation mesh right generated with NetGen

Figure 4.2 shows the original model used, and its tetrahedral representation.

4.1.2 Mesh Partitioning

The simulation mesh is partitioned using an octree data structure—a tree data structure in which each internal node has exactly eight children (see figure 4.3). Octrees are used to partition a three-dimensional space by recursively subdividing the space into eight octants. The reasons for using an accelerated data structure is to avoid performing a brute-force search and using a fast depth-first search instead in order to find the correct tetrahedra—vertex mapping.

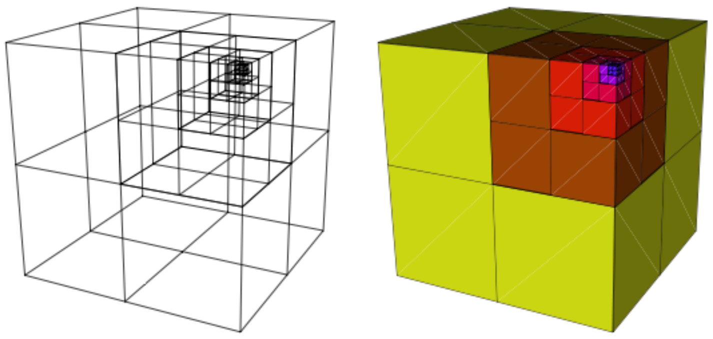


Figure 4.3: An octree representation

4.1.3 Co-rotational Formulation

When an elastic object is subject to a force that deforms its original shape by stretching or compressing it, another internal energy is produced that tries to restore the object to its rest shape. The work, exerted by forces applied on the object, is transformed into strain energy. The definition of elastic energy was given in equation 2.1 in chapter 2. $\boldsymbol{U}$ is the strain energy and $\boldsymbol{W}$ is the work done by external forces, and the relationship can be expressed formalizing the following equation [27]:

$$\begin{aligned} \boldsymbol{W} &= \vec{u_e}^T \vec{f}_{ext} \\ \boldsymbol{U} &= \frac{1}{2}\int_{vol_e} \vec{\epsilon}^T \vec{\sigma} dV \\ \vec{f}_{ext} &= \boldsymbol{K_e} \vec{u_e} \end{aligned}$$

$\vec{f_{ext}}$ are the external forces applied onto the tetrahedron while $\vec{u_e}$ are the displacements of the nodes. $\vec{\epsilon}$ and $\vec{\sigma}$ are respectively the strain and the stress vectors. The relationship between stress and strain is expressed by the following equality known as linear elasticity:

$$\vec{\sigma} = \boldsymbol{D_e}\vec{\epsilon} \tag{4.1}$$

$\boldsymbol{D_e}$ is the elasticity tensor and it relates the Young's modulus and the Poisson's ratio. The strain energy is calculated by integrating strain and stress $\boldsymbol{D_e}$ over the volume (vol_e) of the finite element, i.e., the tetrahedron. The matrix $\boldsymbol{K_e}$ is the stiffness matrix, which represents the system of linear equations that must be solved in order to calculate a solution to the differential equation. $\boldsymbol{K_e}$ can be expressed as follows:

$$\boldsymbol{K_e} = \boldsymbol{S_e^T D_e S_e} \tag{4.2}$$

$\boldsymbol{D_e}$ is the elasticity matrix, which is a 6×6 matrix given by:

$$\boldsymbol{D_e} = \begin{pmatrix} \gamma+\mu & \gamma & \gamma & 0 & 0 & 0 \\ \gamma & \gamma+\mu & \gamma & 0 & 0 & 0 \\ \gamma & \gamma & \gamma+\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu/2 \end{pmatrix}$$

where γ and μ are defined as:

$$\begin{aligned} \gamma &= \frac{1}{36V_e}\frac{\boldsymbol{E}\nu}{(1+\nu)(1-2\nu)} \\ \mu &= \frac{1}{36V_e}\frac{\boldsymbol{E}}{1+\nu} \\ V_e &= \frac{1}{6}b \cdot (c \times d) \end{aligned}$$

b, c and d will be defined shortly. $\boldsymbol{E}$ is the Young's modulus, which is a mechanical property of linear elastic solid materials. It measures the force (per unit area) that is needed to stretch or compress a tetrahedral element and it represents the material's stiffness. The Poisson's ratio ν is related to volume conservation and is the fraction of expansion divided by the fraction of compression. It is usually a dimensionless value between 0 and 0.5 for isotropic materials.

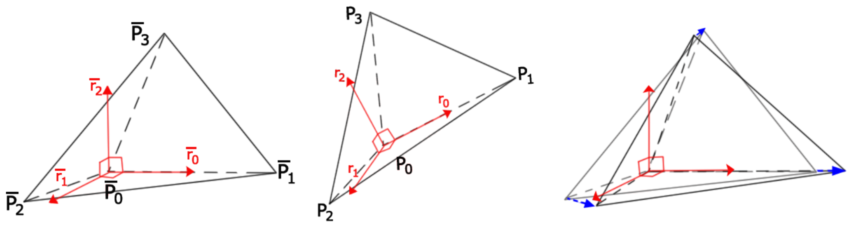


Figure 4.4: Deformations are calculated in a local rotated coordinate system.

$\boldsymbol{S_e}$ is the deformation—displacement matrix. For the co-rotational formulation, this matrix is calculated in the local frame of the element. The linear elasticity formulation is based on the Cauchy strain tensor, which is not rotationally invariant. Thus, for large deformations artefacts and instabilities are introduced into the system, making the simulation inaccurate. (see section 2.6.2. $\boldsymbol{S_e}$ is constant throughout the whole simulation unless the topology of the surface mesh changes, e.g., in case of fractures. Since this implementations does not take into account fractures, $\boldsymbol{S_e}$ can indeed be considered constant. This assumption is possible because of the adoption of linear elastic materials. The displacement vectors are always calculated using the rest position of the tetrahedra and not the history of displacements. $\boldsymbol{S_e}$ can be formulated as follows [27]:

$$\boldsymbol{S_e} = \begin{pmatrix} -p_x(bcd) & 0 & 0 & -p_y(bcd) & 0 & -p_z(bcd) \\ 0 & -p_y(bcd) & 0 & -p_x(bcd) & -p_z(bcd) & 0 \\ 0 & 0 & -p_z(bcd) & 0 & -p_y(bcd) & -p_x(bcd) \\ p_x(cda) & 0 & 0 & p_y(cda) & 0 & p_z(cda) \\ 0 & p_y(cda) & 0 & p_x(cda) & p_z(cda) & 0 \\ 0 & 0 & p_z(cda) & 0 & p_y(cda) & p_x(cda) \\ -p_x(dab) & 0 & 0 & -p_y(dab) & 0 & -p_z(dab) \\ 0 & -p_y(dab) & 0 & -p_x(dab) & -p_z(dab) & 0 \\ 0 & 0 & -p_z(dab) & 0 & -p_y(dab) & -p_x(dab) \\ p_x(abc) & 0 & 0 & p_y(abc) & 0 & p_z(abc) \\ 0 & p_y(abc) & 0 & p_x(abc) & p_z(abc) & 0 \\ 0 & 0 & p_z(abc) & 0 & p_y(abc) & p_x(abc) \end{pmatrix}$$

where $p(uvw) = u \times v + v \times w + w \times u$.

The elements a, b, c, d are the precomputed un-deformed vertex positions in the rotated frame:

$$\begin{aligned} a &= \boldsymbol{R_e}^T(p_0 - p_0) = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T \\ b &= \boldsymbol{R_e}^T(p_1 - p_0) = \begin{pmatrix} b_x & 0 & 0 \end{pmatrix}^T \\ c &= \boldsymbol{R_e}^T(p_2 - p_0) = \begin{pmatrix} c_x & c_y & 0 \end{pmatrix}^T \\ d &= \boldsymbol{R_e}^T(p_3 - p_0) = \begin{pmatrix} d_x & d_y & d_z \end{pmatrix}^T \end{aligned}$$

The rotation matrix $\boldsymbol{R_e} = [r_0 \; r_1 \; r_2]$, which is used to calculate the vertices in the rotated frame, is defined as:

$$
\begin{aligned}
r_0 &= \frac{p_1 - p_0}{\| p_1 - p_0 \|} \\
r_2 &= \frac{(p_1 - p_0) \times (p_2 - p_0)}{\| (p_1 - p_0) \times (p_2 - p_0) \|} \\
r_1 &= r_2 \times r_0
\end{aligned}
$$

4.1.4 Compute Forces

In this step, forces applied to each element and a change in the force, due to vertex displacements, are calculated. Using the co-rotational formulation, introduced in the previous section, expanding $\vec{f_{ext}}$ and $\frac{d\vec{f_{ext}}}{dt}$ leads to:

$$
\begin{aligned}
\vec{f_e} &= \boldsymbol{R_e S_e^T D_e S_e} (\boldsymbol{R_e^T} \vec{u} - \vec{u_0}) \\
\frac{d\vec{f_e}}{dt} &= \boldsymbol{R_e S_e^T D_e S_e R_e^T} \vec{u_e}
\end{aligned}
$$

$\vec{u}$ and $\vec{u_0}$ are, respectively, the displacement in the deformed and the un-deformed state. This formulation is not sufficient to solve the dynamic deformation of the tetrahedra. The differential equation 4.3 is incomplete and thus far from being solved.

$$
\boldsymbol{M}\ddot{\vec{u}} + \boldsymbol{D}\dot{\vec{u}} + \boldsymbol{K}\vec{u} \tag{4.3}
$$

$\boldsymbol{M}$ is the mass and $\boldsymbol{D}$ is the damping; both matrices are constant. K , the stiffness matrix is the same as in equation 4.2. The vector $\vec{u}$, as before, is the difference between the deformed and the rest position $\vec{u} - \vec{u_0}$. It is possible to solve equation 4.3 using several integration techniques. The most viable is the implicit integration scheme. Although the explicit Euler integration is faster than the implicit scheme, it is also less robust and it suffers from conditional stability. Thus, there exists a critical time step size beyond which numerical instabilities appear. The implicit integration scheme adopted will be explained in detail in section 4.1.5.

4.1.5 Integration Method

The implicit backward Euler's integration method provides a faster numerical solution and is more robust than explicit methods. It can be formulated as follows:

$$\vec{x}(t+h) = \vec{x}(t) + h\ \vec{v}(t+h) \tag{4.4}$$

h is the time-step of the physics simulation. For consistency the simulation has a fixed time step of 0.04 seconds. This value has been made adjustable in the simulation parameters. The adoption of different time-steps for physics calculation and for rendering updates is usually a good practice: it gives a coherent behaviour on different platforms where the CPU's clock speed is different. The time between frames is variable and in some cases, especially on slower hardware, it might lead to large time steps, introducing instabilities and error in the simulation.

The position at time $t + h$, using the implicit formulation, can be computed from the velocity at the same time step $t + h$. Equation 4.3 can be rewritten as follows:

$$\begin{aligned}
\boldsymbol{M}\ddot{\vec{v}}(t+h) &= \boldsymbol{M}\vec{v} + h\ \boldsymbol{M}\dot{\vec{v}}(t+h) \\
\boldsymbol{M}\ddot{\vec{v}}(t+h) &= \boldsymbol{M}\vec{v} + h\ [-\boldsymbol{D}\vec{v}(t+h) - \boldsymbol{K}(\vec{u}(t+h) - \vec{u_0}) + f_{ext})]
\end{aligned}$$

Thus, by substituting the first equation into the second, the final linear system of equations is obtained :

$$[\boldsymbol{M} + h\boldsymbol{D} + h^2\boldsymbol{K}]\vec{v}(t+h) = \boldsymbol{M}\vec{v} + h\ [\boldsymbol{K}(\vec{u}(t) - \vec{u_0}) - f_{ext})] \tag{4.5}$$

In order to solve for $\vec{v}(t+h)$ in equation 4.5, the matrix $[\boldsymbol{M} + h\boldsymbol{D} + h^2\boldsymbol{K}]$ must be inverted, and both side of the equation must be multiplied by this inverse. The linear system obtained must be solved at every step of the simulation.

4.1.6 Conjugate Gradient

The linear system in equation 4.5 is in the form $\boldsymbol{A}\vec{x} = \vec{b}$, where the matrix A is defined as symmetric, positive definite and sparse. In this case the conjugate gradient (CG) method can be adopted. Conjugate gradient performs well if the matrix is sparse, i.e., having most of its elements set to zero. This speeds up matrix—matrix and matrix—vector multiplications considerably. However, under some conditions, the inversion of a sparse matrix can result in a dense matrix requiring more memory and computational time. The conjugate gradient is an iterative algorithm, where a given initial solution is refined until an optimum is reached. It runs in N iterations, where N is the dimension of $\boldsymbol{A}$. For this implementation N is set to 25. As it is part of the simulation parameters, the iteration number can be changed; however, good plausible simulations are achieved using 10 to 25 iterations only. See listing 1 for the pseudo-code of the CG implementation.

```
**Data**: N: number of iterations,
$d = f_0 + hKv$,
$\delta_0 = d \cdot d$: the initial error
**while** $i\ != N$ and $\delta_i > \epsilon^2\delta_0$ **do**
    Calculate delta force:
    $dforce = Kd$;
    $q = Md - h^2 dforce$;
    Calculate error:
    $\alpha = \delta_{i-1}/(d \cdot q)$;
    $a = a + \alpha d$;
    $r = r - \alpha q$;
    $\delta_i = r \cdot r$;
    $\beta = \delta_i/\delta_{i-1}$;
    $d = r + \beta d$;
**end**
```

Algorithm 1: Conjugate gradient pseudo-code

The iterative solver accepts three arguments as input parameters: a number N of iterations, which is chosen during the simulation initialisation; the initial displacement vector product d; and an initial error δ_0, which is calculated as the dot product of d with itself. On every iteration the error is compared with a threshold ϵ to decide whether or not to quit the loop. There are two main steps in the body of the loop: delta force and error calculation. The former is the delta force of the displacement, while the latter combines the data from all objects in order to calculate the error. Forces evaluation and delta force calculation are the most expensive tasks in the algorithm, as will be shown in later in the results section.

4.1.7 Mesh Mapping

The last step performed in the pipeline is the displacement mapping or mesh mapping. The displacement field over the tetrahedron $\vec{u} = [u_x, u_y, u_z]$ can be obtained by linear interpolation of the tetrahedron nodal displacements:

$$\vec{u} = \xi_1 \vec{u}_1 + \xi_2 \vec{u}_2 + \xi_3 \vec{u}_3 + \xi_4 \vec{u}_4 \tag{4.6}$$

where ξ_i are the shape functions and $\vec{u}_i$ are the displacements of the four nodes of the tetrahedron. The associated vertices of each tetrahedron are found by an optimised search using the octree calculated during the simulation initialisation.

4.2 Libraries Used

There are two main libraries used for this project: SOFA framework [37] and NetGen [35]. The former is a framework to run soft dynamics and is frequantly used to simulate medical scenarios. It contains several convenient data structures that maximise cache reuse and helper methods with fast matrix and vector operations. NetGen is a mesh tetrahedralisation tool and was used to generate the simulation mesh of the model used for the experiments.

Chapter 5

Results

In this chapter we present the results of our experiments on two groups of four graphs are shown for each experiment. A group contains the results of the particular architecture targeted and tested. Three of the four graphs compare the FEM functions described in the pipeline in section 4.1. The comparison is performed on two different architectures: i7 and ARM for the CPUs, and GK110A and GK20A for the GPUs. The plot shown on the bottom right of each group of graph compares the average frame rate of the whole algorithm.

The simulation parameters used are listed in table 5. The parameters that affect the speed of the simulation are: time step, CG max iteration and tolerance. The other parameters are mechanical properties of the elastic material and affect the visual aspect of the simulation only.

5.1 Experiment 1

The goal of the first experiment is to benchmark and to evaluate the CPU implementation of the FEM on both desktop and mobile. The desktop CPU reference is an Intel i7 4930k 3.4 GHz with 6 cores and 12 threads. The FEM implementation on the CPU is fully serial: neither multi-threading nor *Single Instruction Multiple Data* (SIMD) instructions are employed. Thus, results relate to a single core.

The two CPUs show similar results in terms of frame rate and time spent per function. By analysing the trends of the mean and the standard deviation it is possible to

Parameter	**Value**
Time step	0.04
CG Max Iteration	25
Rayleigh Stiffness	0.01
Rayleigh Mass	0.01
Tolerance	10^{-3}
Young Modulus	10^{6}
Poisson Ratio	0.4
Gravity	-9.8
Mass Density	0.01
ODE Solver	EulerImplicit

Table 5.1: Simulation parameters.

identify two different behaviours. However, it was not possible to read cache utilisation on the ARM; therefore, cache misses and hits are not given. (The tool available in the NVIDIA development kit for Tegra[1] does not provide this information). As an alternative, *perf* , which is a Linux command-line tool, can be used to read cache utilisation. Unfortunately *perf* is not compatible with Android.

5.1.1 CPU Comparison

Figure 5.1 shows the time spent, in milliseconds per frame, in the three main functions of the FEM algorithm. The conjugate gradient, as expected, was the most expensive, taking a total of *27.4 ms* on the ARM and *16.3 ms* on the i7. An average of *1.096 ms* is spent per iteration, where the maximum number of iterations is set to 25 (see table 5). It is interesting to notice that the ARM shows two different trends across all of the graphs, the first before and second after the 450*th* frame. After the 450*th* frame the values become more regular, suggesting better cache reuse. For each core L1 and L2 cache sizes are similar in size and configuration on both Intel and ARM

[1] *Tegra System Profiler*

architectures. The only difference is the presence of a *Snoop Control Unit* (SCU) in the ARM architecture, which maintains data cache coherency between the cores of the CPU.

The frame rate is relatively high on both CPUs, typically 40 to 50 frame per second. Although the ARM shows lower performance, the FEM implementation is more stable in term of peaks and frame rate drops. Table 5.2 summarizes the results.

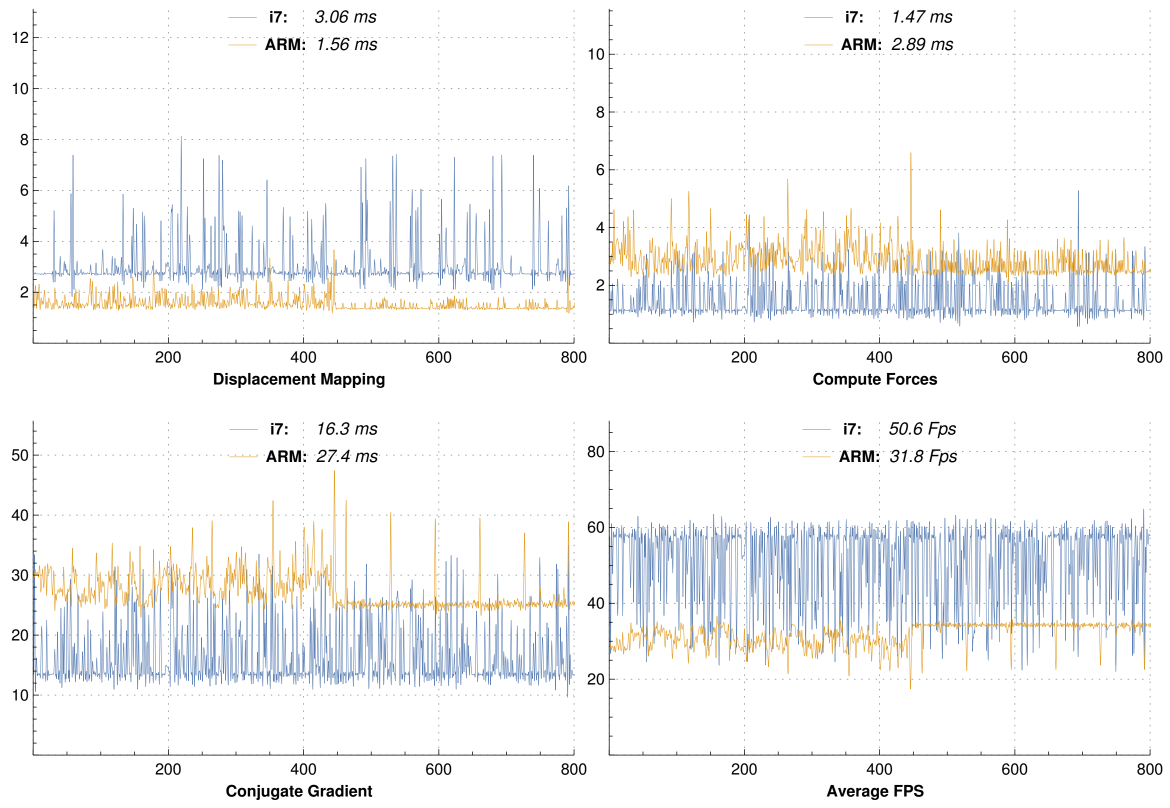


Figure 5.1: CPU FEM Implementation. Performance comparison.

5.1.2 Memory Analysis

In spite of the ARM CPU yielding a lower frame rate in the simulation, it is evident that the conjugate gradient trend is steadier some time after the beginning of the simulation. The lower standard deviation value confirms this result (see table 5.2). The reasons for this difference are unknown and understanding the cause would require a lower-level

	Architecture	**Mean**	**Std. deviation**	**Min**	**Max**
Displacement Mapping	i7	3.06	0.861	2.7	8.13
	ARM	1.51	0.329	1.33	8.84
Compute Forces	i7	1.47	0.612	1.11	5.28
	ARM	2.8	0.478	2.35	6.87
Conjugate Gradient	i7	16.3	4.7	13.4	36.6
	ARM	26.5	2.86	23.1	52.4

Table 5.2: CPU statistics comparison. Values are expressed in milliseconds.

analysis to read cache L1 and L2 hit and misses on the ARM, which was not possible as it requires specific tools.

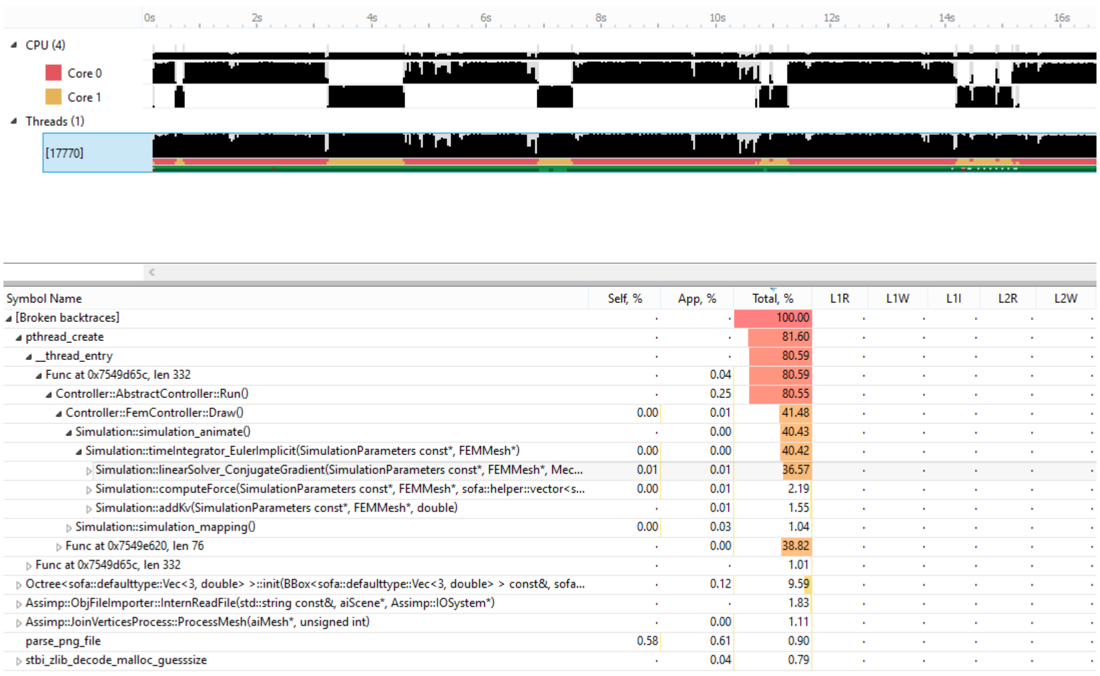
Despite this limitation (see figure 5.2), it was possible to analyse the thread efficiency. The total CPU utilisation is only 25%. Only two out of the four cores available are involved in the computation. This information suggests that there is scope for optimisation on the CPU, as long as the algorithm is re-designed to use multi-core and SIMD instructions.

The current FEM implementation uses only part of the total throughput of the CPU. Without optimising the FEM implementation to support multi-threading a frame rate of approximately 42 FPS was successfully achieved. It is remarkable that maximum power consumption of only 0.35W was used to produce this result.

5.2 Experiment 2

The second experiment performs the same tests as in experiment 1. The simulation parameters and conditions are the same as those of experiment 1. The reference GPU used is the GeForce 780ti (GK110A), which employs the same Kepler architecture as the Shield Tegra (GK20A). Differences between the two GPUs are highlighted in chapter 3.2. In summary, the two architectures are very similar but with some differences:

- The GK110A has 15 SMXs for a total of 2880 SPs while the GK20A has only one SMX with 192 SPs.



Symbol Name	Self, %	App, %	Total, %	L1R	L1W	L1I	L2R	L2W
◢ [Broken backtraces]	·	·	100.00	·	·	·	·	·
◢ pthread_create	·	·	81.60	·	·	·	·	·
◢ __thread_entry	·	·	80.59	·	·	·	·	·
◢ Func at 0x7549d65c, len 332	·	0.04	80.59	·	·	·	·	·
◢ Controller::AbstractController::Run()	·	0.25	80.55	·	·	·	·	·
◢ Controller::FemController::Draw()	0.00	0.01	41.48	·	·	·	·	·
◢ Simulation::simulation_animate()	·	0.00	40.43	·	·	·	·	·
◢ Simulation::timeIntegrator_EulerImplicit(SimulationParameters const*, FEMMesh*)	0.00	0.00	40.42	·	·	·	·	·
▷ Simulation::linearSolver_ConjugateGradient(SimulationParameters const*, FEMMesh*, Mec...	0.01	0.01	36.57	·	·	·	·	·
▷ Simulation::computeForce(SimulationParameters const*, FEMMesh*, sofa::helper::vector<s...	0.00	0.01	2.19	·	·	·	·	·
▷ Simulation::addKv(SimulationParameters const*, FEMMesh*, double)	·	0.01	1.55	·	·	·	·	·
▷ Simulation::simulation_mapping()	0.00	0.03	1.04	·	·	·	·	·
▷ Func at 0x7549e620, len 76	·	0.00	38.82	·	·	·	·	·
▷ Func at 0x7549d65c, len 332	·	·	1.01	·	·	·	·	·
▷ Octree<sofa::defaulttype::Vec<3, double> >::init(BBox<sofa::defaulttype::Vec<3, double> > const&, sofa...	·	0.12	9.59	·	·	·	·	·
▷ Assimp::ObjFileImporter::InternReadFile(std::string const&, aiScene*, Assimp::IOSystem*)	·	·	1.83	·	·	·	·	·
▷ Assimp::JoinVerticesProcess::ProcessMesh(aiMesh*, unsigned int)	·	0.00	1.11	·	·	·	·	·
parse_png_file	0.58	0.61	0.90	·	·	·	·	·
▷ stbi_zlib_decode_malloc_guesssize	·	0.04	0.79	·	·	·	·	·

Figure 5.2: CPU thread and core utilisation

- The GK20A employs a unified memory architecture, which means that the memory is shared between GPU and CPU.

Unfortunately there was insufficient time to adapt the algorithm to four different architectures (CPU and GPU). Thus, the GPU Shield implementation will not benefit from the usage of unified shared memory.

5.2.1 GPU Comparison

Figure 5.3 shows the performance comparison of the FEM implementation on the GPU for both architectures, desktop and mobile. Unexpectedly, the GPU Tegra implementation shows no major improvement, performing an average of just 40 FPS. The displacement mapping graph on the top left in figure 5.3 shows similar timings to the same graph in figure 5.1. This is evidence of inefficiency in the CUDA implementation, which will be addressed in the memory analysis in section 5.2.2.

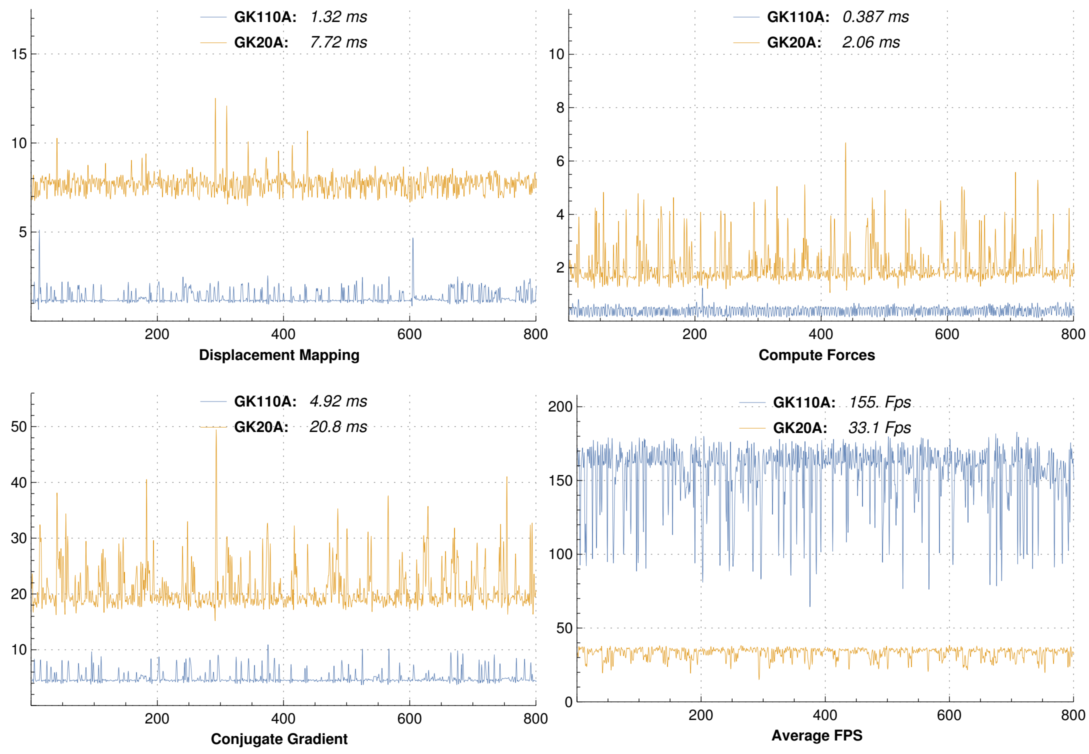


Figure 5.3: GPU FEM Implementation. Performance comparison.

5.2.2 Memory Analysis

To address the performance problems mentioned in the previous section, tests were performed on the memory hierarchy to analyse bottlenecks, warp and memory efficiency issues. For this analysis two tools were used:

- NSight Visual Studio profiler
- Nvprof

The former is a graphical profiling tool, which we used to analyse and profile data from the CUDA implementation run on the GK110A. As of now there are no graphical tools to profile a CUDA application from an APK, i.e., a package file format used to distribute and install application software onto Google's Android operating system. Nvprof is a command-line tool and it was used to record profiling sessions on the Shield.

	Architecture	Mean	Std. Deviation	Min	Max
Displacement Mapping	GK110A	1.33	0.341	1.12	5.12
	GK20A	7.78	0.505	6.83	12.5
Compute Forces	GK110A	0.386	0.155	0.185	2.15
	GK20A	2.1	0.811	1.06	15.2
Conjugate Gradient	GK110A	4.96	1.04	4.25	10.5
	GK20A	21.2	4.09	17.1	56.9

Table 5.3: GPU statistics comparison. Values are expressed in milliseconds.

	Dimension				**Allocated per Block**				
Kernel	Grid	Blocks	Duration(μs)	Registers per Thread	Warps	Registers	S. Memory	% Achieved Occupancy	% Branch Efficiency
CalculateForce	188	64	19.488	15	2	1024	768 B	0.233	0.821
CalculateDForce	132	64	8.288	42	2	3072	3328 KB	0.248	1
Plane_AddForce	47	64	5.056	10	2	1024	768 B	0.093	1
Plane_AddDForce	47	64	4.928	11	2	1024	768 B	0.093	1
AddMDx	141	64	4.032	8	2	512	0	0.244	1
Calculate_vDot	71	128	3.968	10	4	2048	512 B	0.240	0.999

Table 5.4: GK110A results

GK110A

Table 5.4 summarizes the results of the experiments carried out in running the kernels in the CUDA FEM implementation. In the table, we show the grid and block dimensions. The grid number is determined using the following formula:

$$threadsPerBlock = 64$$
$$gridSize = (inputSize + threadsPerBlock - 1)/threadsPerBlock$$

Blocks use the global constant `threadsPerBlock`. For this experiment it was set to 64 threads per block. For each function the average duration in picoseconds is reported. A number of registers per thread are used to store local variables and function argument values. Table 5.4 also shows the total allocated warps, registers and shared memory per block. In addition to these values, we reported values to determine the warp efficiency: *achieved occupancy* and *branch efficiency*. Warp occupancy is defined as the ratio of active warps on an streaming multiprocessor to the maximum number of active warps supported by the SM. Occupancy varies over time as warps begin and end, and can be different for each SM [38]. During the time threads in a warp begin executing to the time when all threads in the warp have exited from the kernel, a warp is considered active. Achieved occupancy is very low overall, approximately 25%, which translates to circa 14 active warps out of the theoretical total 32 per block. Low occupancy indicates instruction issue efficiency, because there are not enough eligible warps to hide latency between dependent instructions [38]. Figure 5.4 provides an insight into these inefficiencies and it highlights where stalls are generated. As is evident from graph c) the main cause for low occupancy and warp efficiency is memory dependency. Memory dependency stalls are caused by required resources not being available or fully utilised. This is a dependency on data being loaded from memory which has not yet arrived, and indicates that performance is being limited due to memory latency. It is also clear that the cache is not fully utilised. Figure 5.5 shows the memory statistics for the function *CalculateForce*.

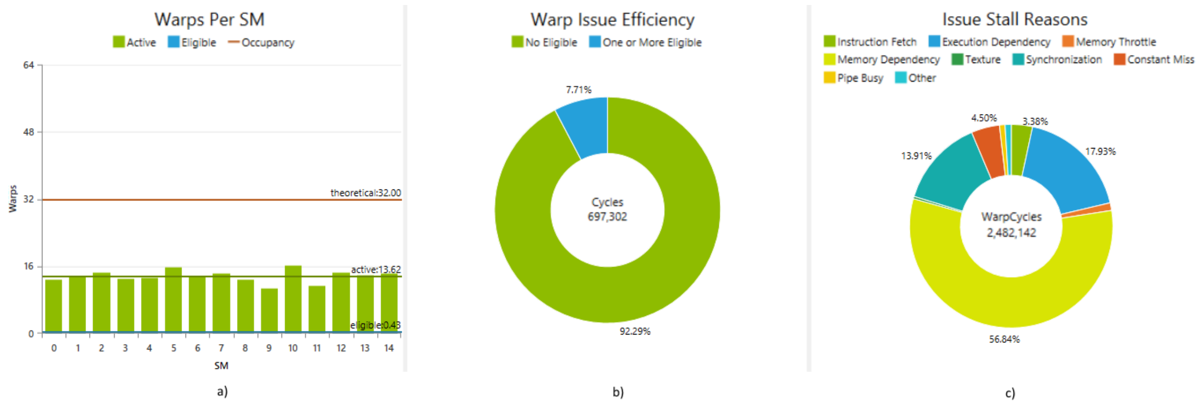


Figure 5.4: Warp issue efficiency. a) Number of active warps per SM, b) eligible warps per cycle and c) causes of warp stall

Surprisingly, the L1 cache has no hits while the greatest accesses are to the L2 cache, shared and device (global) memory. This explains the reason to memory dependency:

latencies to retrieve data from shared and global memory are too high. Thus, a warp must wait until the memory becomes available for each of its threads. Stores and loads between L1 and L2 cache are shown in figure 5.6. L2 utilisation is good overall (75%) but again L1 is not hit at all. These aspects can be mitigated by optimizing memory alignment and access patterns.

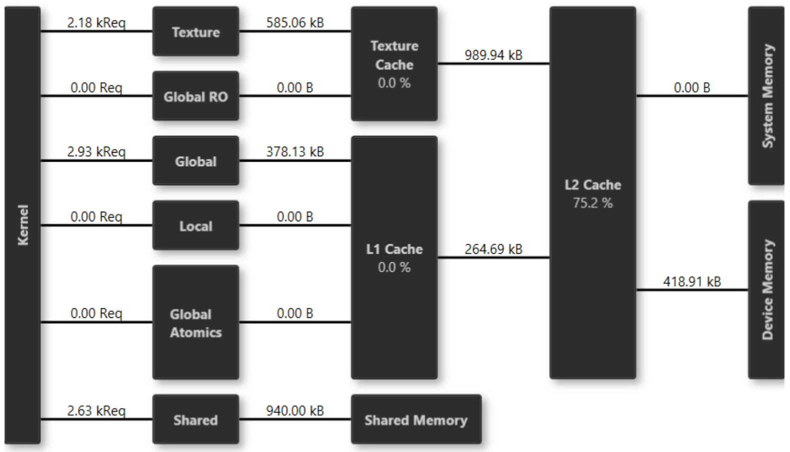


Figure 5.5: Memory statistics.

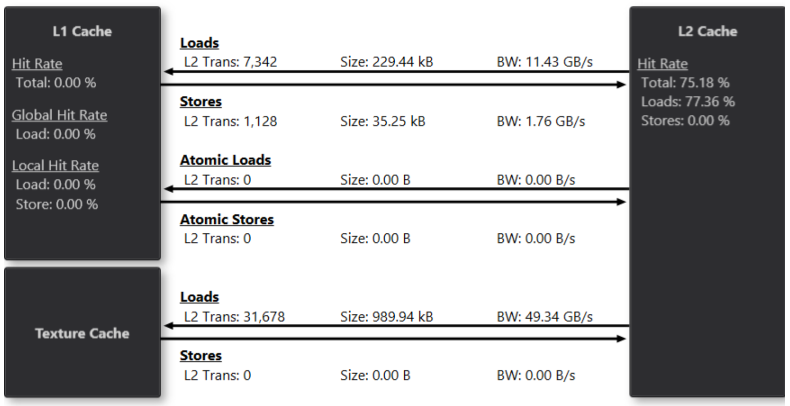


Figure 5.6: Cache utilisation.

Another important aspect of the analysis is branch efficiency, which measures the ratio between flow control decisions over all executed branch instructions. Branch divergence occurs when threads of a warp are forced to take different executions path. If this happens, the different execution paths must be serialised, since all of the threads of a warp share a program counter; this increases the total number of instructions executed for the warp [38]. In this case, the diverged threads are put to sleep until they converge back to the original branch.

Branching is caused by loops and flow control instructions. However, almost all of the kernels in table 5.4 achieve 100% branch efficiency. *CalculateForce*, on the other hand, has a lower value of 82.1%.

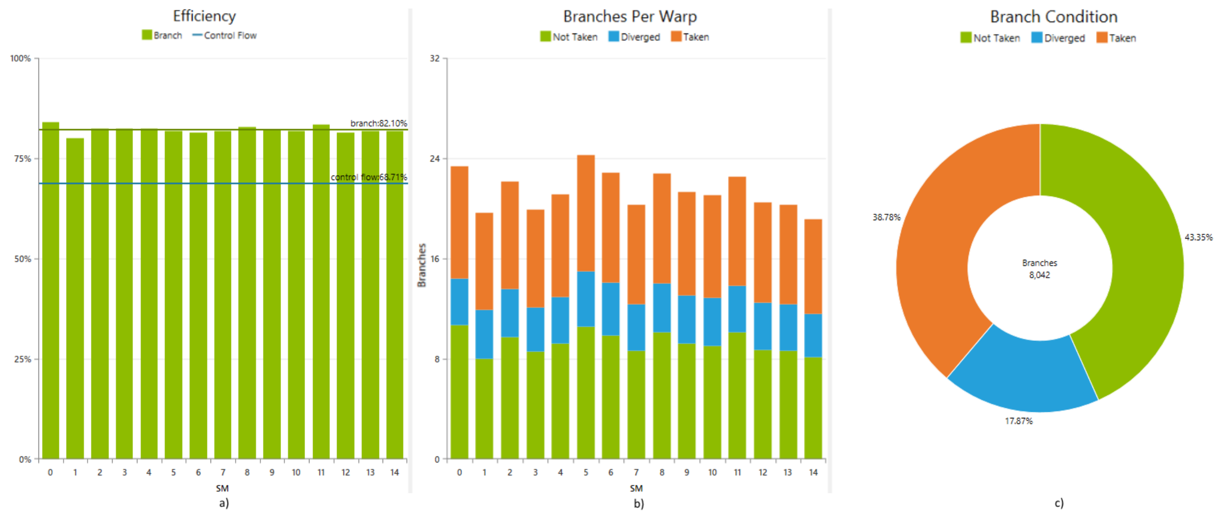


Figure 5.7: Branch efficiency

Figure 5.7 shows the branch efficiency statistics. Graph a) shows two metrics for evaluating the impact of flow control: branch and flow control efficiency. The former represents branch efficiency per SM (the bars), while the latter represents an average over all SMs (the branch line). Higher values are better, and indicate that warps take a uniform execution path. The kernel *CalculateForce* has a good branch efficiency of 82.10%, as shown in graph a) of figure 5.7. Control flow is worse at only 68.71%.

Graph b) shows the average count of executed branch instructions per warp per SM. It contains three metrics: not taken, taken and diverged. *Taken* and *not taken* are the average number of executed branch instructions with a uniform control flow decision per warp, i.e., the active threads of a warp that either take or do not take the branch. *Diverged* is the average number of executed branch instructions per warp for which the conditional resulted in different outcomes across the threads of the warp [38]. This graph explains why the kernel has a low control flow efficiency (see graph

	Dimension				**Allocated per Block**				
Kernel	Grid	Block	Duration (μs)	Register per thread	Warps	Registers	S. Memory	% Achieved Occupancy	% Branch Efficiency
CalculateForce	188	64	148.511	13	2	1024	768 B	0.468	0.624
CalculateDForce	188	64	112.701	38	2	2560	3.25 KB	0.417	1
Plane_AddForce	47	64	20.791	12	2	1024	768 B	0.454	1
Plane_AddDForce	47	64	18.199	11	2	1024	768 B	0.449	1
AddMDx	141	64	15.783	8	2	512	0 B	0.441	1
Calculate_vDot	71	128	16.035	9	2	2048	512 B	0.712	1

Table 5.5: GK20A results

a). Higher values of *taken*, with lower values of *not taken* and *diverged*, indicate that the warp is not branching or stalling by taking different execution paths.

GK20A

Table 5.5 shows the results obtained from profiling the CUDA kernel using the GPU GK20A on the Tegra device. The same kernel functions of table 5.4 are listed, as they proved to be computationally the most expensive. As expected, the overall execution time, per kernel, is higher compared to its desktop reference. However, it is of more interest to study resource utilisation, warp and instruction efficiency. The function *CalculateForce*, which is the most expensive kernel, is taken as a reference to analyse warp and instruction efficiency, resource utilisation, and memory bandwidth and utilisation.

Surprisingly, the achieved occupancy is approximately 48%, which is 25% higher than the device GK110A. However, occupancy is still not optimal as there remains 65% of the warp not being utilised. The kernel has a block size of 64 threads. This size is likely preventing the kernel from fully utilising the GPU. The device can simultaneously execute up to 16 blocks on each SM. Because each block uses 2 warps to execute the block's 64 threads, the kernel is using only 32 warps on each SM. Increasing the number of threads in each block can increase the amount of warps that can execute on each SM. However, increasing the occupancy in a kernel that is subject to branch divergence would also increase the number of divergent threads, raising the overall branching inefficiency. In fact, branch efficiency reported in table 5.5 is 20% lower than its corresponding execution on the GK110 device (see table 5.4).

The kernel *ComputeForce* exhibits low compute throughput and memory bandwidth utilisation relative to the peak performance of the GK20A device. These utilisation levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues. (see figure 5.8).

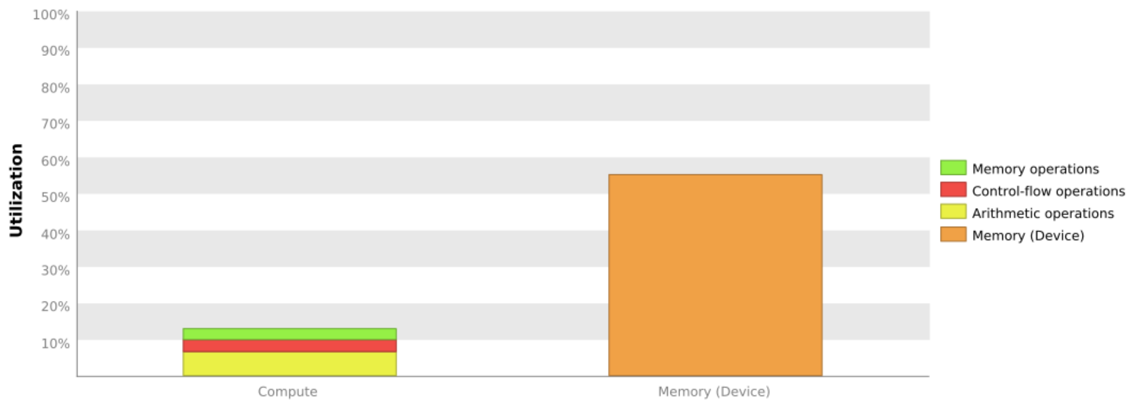
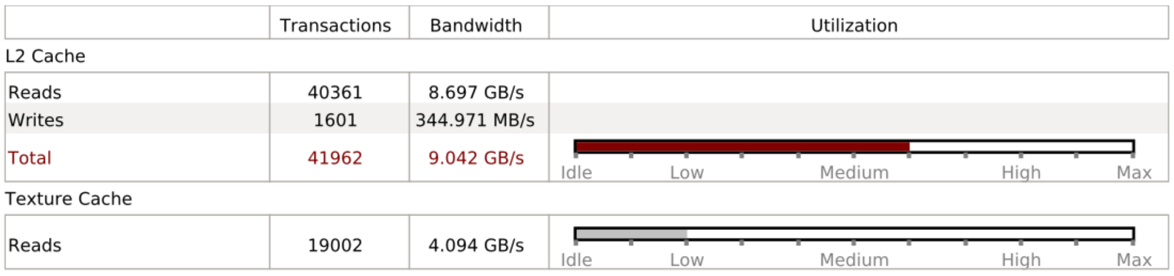


Figure 5.8: Branch efficiency

The result shown in figure 5.8 reveals the same memory dependency issue, which is the cause of stalls. The kernel's warp execution efficiency of 62.4% is less than 100% due to divergent branches and control flow instructions.



	Transactions	Bandwidth	Utilization
L2 Cache			
Reads	40361	8.697 GB/s	
Writes	1601	344.971 MB/s	
Total	41962	9.042 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	19002	4.094 GB/s	Idle Low Medium High Max

Figure 5.9: Memory bandwidth efficiency

Figure 5.9 shows the memory bandwidth used by the kernel for the various types of memory on the device. The table also shows the utilisation of each memory type relative to the maximum throughput supported by the memory. The result shows that the kernel is limited by the bandwidth available to the L2 cache. Unfortunately, the profiling tool on Android does not provide any information about L1 cache utilisation. Register pressure can prevent the kernel from fully utilising the GPU. However, this kernel does not exhibit much register usage, as only 1024 registers are allocated per block (13 allocated registers per thread).

Chapter 6

Conclusions

The main goal of developing an interactive FEM implementation has been achieved with all of the four experiments. A cross-platform OpenGL ES 3.0 FEM was developed using CUDA (compute capability 3.2) for parallel calculations on the GPU.

Results were obtained and compared on four different architectures: Intel i7 4930k and ARM Cortex A15 for the CPUs and Kepler architecture, desktop and mobile, for the GPUs.

The ARM CPU single core implementation on the Tegra exhibited similar performances to the GPU implementation. The CPU converges to a steadier frame rate some time after the beginning of the simulation. Unfortunately the causes of this behaviour are unknown, but the presence of a *Snoop Control Unit* in the ARM processors, which guarantees cache-to-cache data transfer between cores, is a plausible explanation. However, a lower-level analysis is required to proof this hypothesis because it is necessary to read cache hits and misses on the ARM.

An analysis on the memory hierarchy of the GPU revealed that warp and branching inefficiencies are the main bottlenecks for the simulation. These limitations are addressed in chapter 5, and causes are proved to be in thread-to-thread memory dependencies and low cache reuse. Moreover, unified memory architecture was not fully exploited in the provided CUDA implementation, creating a duplication of the memory footprint.

The ARM CPU is better suited for running soft-body deformation on mobile platforms. The ARM exhibited lower performance but higher stability. This makes the

CPU ideal for running visually plausible physics simulations. Moreover, CPU reached only 25% of usage during the whole simulation, which is a large margin to run other aspects of a system (e.g., AI, collision detection and response with other rigid/soft objects, etc.).

6.1 Future Work

The use of the conjugate gradient is computationally expensive. There exists other solvers for finding solutions to the linear system of equation in the FEM, e.g., direct methods, Cholesky factorization, etc. These solvers could be implemented to study the impact they have on performance and on the memory hierarchy.

Fractures could be an interesting addition as well as adding more physical objects to the scene, in order to perform collision detection and calculate responses.

A lower-level analysis of the ARM CPU cache could be performed. Unfortunately, the Shield Tegra employs an Android operating system, which is a limited version of the Linux kernel. Thus, tools to analyse the memory hierarchy, e.g., *perf*, are not available. The embedded board *Jetson TK1* could be a good alternative because is based on the Kepler architecture.

Using unified memory on the Shield could improve the overall performance and address the memory dependency and efficiency issues reported in chapter 5.

Bibliography

[1] NVIDIA, "Nvidia® kepler gk110," pp. 1–24, NVIDIA Corporation, 2013.

[2] NVIDIA, "Nvidia® tegra® k1," pp. 1–26, NVIDIA Corporation, 2014.

[3] NVIDIA, "Cuda c programming guide," 2015.

[4] M. Müller and M. Gross, "Interactive virtual materials," in *Proceedings of Graphics Interface 2004*, GI '04, (School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada), pp. 239–246, Canadian Human-Computer Communications Society, 2004.

[5] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, and B. Cutler, "Stable real-time deformations," in *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 49–54, ACM, 2002.

[6] M. Nesme, Y. Payan, and F. Faure, "Efficient, physically plausible finite elements," in *Eurographics 2005, Short papers, August, 2005* (J. Dingliana and F. Ganovelli, eds.), (Trinity College, Dublin, Irlande), 2005.

[7] E. G. Parker and J. F. O'Brien, "Real-time deformation and fracture in a game environment," in *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, (New York, NY, USA), pp. 165–175, ACM, 2009.

[8] N. B. J. Andrews, "Xbox360 architecture," pp. 25–37, IEEE Computer Society, 2006. 0272-1732/06/20.00.

[9] M. Müller, N. Chentanez, and T.-Y. Kim, "Real time dynamic fracture with volumetric approximate convex decompositions," *ACM Trans. Graph.*, pp. 115–115, 2013.

[10] A. Nikolov, "Volume rendering optimisations for mobile devices," Master's thesis, Trinity College of Dublin, 2015.

[11] S. F. F. Gibson and B. Mirtich, "A survey of deformable modeling in computer graphics," tech. rep., 1997.

[12] J. Mesit, *Modeling and Simulation of Soft Bodies.* PhD thesis, University of Central Florida Orlando, Florida, 2010.

[13] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, "Position based dynamics," *J. Vis. Comun. Image Represent.*, vol. 18, pp. 109–118, Apr. 2007.

[14] X. P. Institut and X. Provot, "Deformation constraints in a mass-spring model to describe rigid cloth behavior," in *In Graphics Interface*, pp. 147–154, 1996.

[15] C. Garre and A. Pérez, "A simple mass-spring system for character animation,"

[16] S. Xu, X. Liu, H. Zhang, and L. Hu, "An improved realistic mass-spring model for surgery simulation," in *Haptic Audio-Visual Environments and Games (HAVE), 2010 IEEE International Symposium on*, pp. 1–6, Oct 2010.

[17] T. Liu, A. W. Bargteil, J. F. O'Brien, and L. Kavan, "Fast simulation of mass-spring systems," *ACM Trans. Graph.*, vol. 32, pp. 214:1–214:7, Nov. 2013.

[18] M. Müller, "Hierarchical position based dynamics," 2008.

[19] J. Bender, M. Müller, M. A. Otaduy, and M. Teschner, "Position-based methods for the simulation of solid objects in computer graphics," *EUROGRAPHICS 2013 State of the Art Reports*, 2013.

[20] Y. Wang, Y. Xiong, K. Xu, K. Tan, and G. Guo, "A mass-spring model for surface mesh deformation based on shape matching.," in *GRAPHITE*, vol. 6, pp. 375–380, 2006.

[21] W. Rungjiratananon, Y. Kanamori, and T. Nishita, "Chain shape matching for simulating complex hairstyles," in *Computer graphics forum*, vol. 29, pp. 2438–2446, Wiley Online Library, 2010.

[22] A. R. Rivers and D. L. James, "Fastlsm: fast lattice shape matching for robust real-time deformation," in *ACM Transactions on Graphics (TOG)*, vol. 26, p. 82, ACM, 2007.

[23] S. Bouaziz, S. Martin, T. Liu, L. Kavan, and M. Pauly, "Projective dynamics: fusing constraint projections for fast simulation," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, p. 154, 2014.

[24] J. F. O'Brien and J. K. Hodgins, "Graphical modeling and animation of brittle fracture," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, (New York, NY, USA), pp. 137–146, ACM Press/Addison-Wesley Publishing Co., 1999.

[25] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, "Elastically deformable models," *SIGGRAPH Comput. Graph.*, vol. 21, pp. 205–214, Aug. 1987.

[26] J. F. O'Brien, A. W. Bargteil, and J. K. Hodgins, "Graphical modeling and animation of ductile fracture," *ACM Trans. Graph.*, vol. 21, pp. 291–294, July 2002.

[27] J. Allard, H. Courtecuisse, and F. Faure, "Implicit FEM Solver on GPU for Interactive Deformation Simulation," in *GPU Computing Gems Jade Edition* (W. mei W. Hwu, ed.), Applications of GPU Computing Series, pp. 281–294, Elsevier, Nov. 2011.

[28] S. Fortune, "A sweepline algorithm for voronoi diagrams," in *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, (New York, NY, USA), pp. 313–322, ACM, 1986.

[29] H. Ledoux, "Computing the 3d voronoi diagram robustly: An easy explanation," in *Voronoi Diagrams in Science and Engineering, 2007. ISVD '07. 4th International Symposium on*, pp. 117–129, July 2007.

[30] *FEM Simulation of 3D Deformable Solids: A practitioner's guide to theory, discretization and model reduction. Part One: The classical FEM method and discretization methodology. SIGGRAPH '12: ACM SIGGRAPH 2012 Courses.*, 2012.

[31] C. Felippa and B. Haugen, "A unified formulation of small-strain corotational finite elements: I. theory," *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 21–24, pp. 2285 – 2335, 2005. Computational Methods for Shells.

[32] J. Georgii and R. Westermann, "Corotated Finite Elements Made Fast and Stable," pp. 11–19, 2008.

[33] ARM, "Arm® architecture reference manual - armv7-a and armv7-r edition," pp. 1–2736, ARM, 2015.

[34] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE micro*, no. 2, pp. 39–55, 2008.

[35] N. Arnold, "Netgen/ngsolve manual," 2013.

[36] *Game physics pearls*, ch. Soft Bodies Using Finite Elements, pp. 217–248. Natick, Mass: A.K. Peters, 2010.

[37] J. Allard, S. Cotin, F. Faure, P. j. Bensoussan, F. Poyer, C. Duriez, H. Delingette, and L. G. B, "Sofa – an open source framework for medical simulation," in *In Medicine Meets Virtual Reality (MMVR 15*, 2007.

[38] NVIDIA, "Nvidia nsight visual studio edition user guide," 2015.