A Study of Deformation and Fracture in Virtual Environments

by

Daniel Walsh, BAI

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

M.Sc. in Computer Science

(Interactive Entertainment Technology)

University of Dublin, Trinity College

August 2016

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.

____

Daniel Walsh

August 30, 2016

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Daniel Walsh

August 30, 2016

A Study of Deformation and Fracture in Virtual Environments

Daniel Walsh
University of Dublin, Trinity College, 2016

Supervisor: Michael Manzke

Real-time deformation and fracture have been an open area of research for over two decades, with various approaches being explored in this time.

One of the fastest, and most stable methods, shape matching is a simplified, position-based physics model, originally proposed over ten years ago that has proven viability in soft-body simulation that is also extensible to support fracturing of objects.

In this dissertation, I have taken an existing shape matching algorithm supporting ductile fracture and developed a highly parallel implementation that runs on either CPU or GPU, and performed a thorough performance analysis to evaluate the solution and guide future work.

Contents

Abstract **iv**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 2

Chapter 2 Background Research **3**

2.1 State of the Art 3

2.1.1 Voronoi Fracture 3

2.1.2 Mass-Spring Systems 4

2.1.3 Finite Element Method 5

2.1.4 Position Based Dynamics 6

2.1.5 Clustered Shape Matching For Ductile Fracture 8

2.2 General Purpose GPU Programming 11

2.3 OpenCL Architecture 11

Chapter 3 Experimental Design **14**

3.1 Target Platforms 14

3.2 Measurements 15

3.2.1 Frame Time 16

3.2.2 Hot Spots 16

3.2.3 Memory Analysis 16

Chapter 4 Implementation **17**

4.1 OpenGL Renderer 17

4.2 OpenCL Optimisation 17
4.3 Parallel Physics 18
4.4 Shortcomings 21

Chapter 5 Results **23**
5.1 Frame Times 24
5.1.1 Test Machine 1 24
5.1.2 Test Machine 2 26
5.2 Hotspot and Memory Analysis 27

Chapter 6 Conclusions and Future Work **31**
6.1 Conclusions 31
6.2 Future Work 32

Bibliography **41**

Chapter 1

Introduction

1.1 Motivation

Modern real-time graphical applications use state-of-the-art effects to improve visuals to the point that many achieve near photorealism, but environments are still largely static, with deformation and fracture either very limited in their use, or not present at all. This leads to a significant disconnect where applications have highly realistic visuals but without the physical plausibility to make the environment truly believable.

There have been many approaches used in academia to solve this problem and brittle fracture has been shown to be achievable at real-time speeds in a variety of ways [20] [22] [14] but soft-body simulation and ductile fracture remain out of scope of most real-time applications. Though in many cases, these physics simulations may be run at real-time speeds[16][11][12], they are typically still too costly to be integrated into a larger interactive application.

Due to the failure of Dennard scaling, there has been a general trend toward multi-core architectures since 2005, creating an increased demand for more highly threaded software. In a similar vein, recent CPU generations have seen only minor incremental performance improvements while GPUs continue to show considerable performance increases between generations. Due to these facts it is becoming increasingly important to explore multi-threaded implementations of computationally expensive algorithms, with particular emphasis on highly parallel GPU implementations.

Clearly then, exploiting the increased parallelism of modern hardware may allow

for significant performance improvements in many previously explored approaches to modeling deformation and fracture in a real-time setting, allowing for far richer real-time simulations.

1.2 Objectives

The main objective of this dissertation is to expand upon the work of Jones et. al[11], by creating a highly parallel GPU implementation of their clustered shape matching algorithm, that supports soft-body simulation and ductile fracture.

Profiling this parallelised implementation and comparing it to the baseline performance of the CPU implementation will prove whether or not it may provide significant performance improvement for this algorithm and guide future work in the area.

Since brittle fracture may be modeled as a special case of ductile fracture (with extremely high material toughness), an efficient enough ductile fracture algorithm may also be used to simulate brittle fracture, producing a complete solution to handle both brittle and ductile fracture at real-time speeds would allow the simulation of a far greater range of physical phenomena and greatly improve the realism and believability of modern real-time applications.

Chapter 2

Background Research

2.1 State of the Art

The simulation of fracture mechanics in 3D graphics has been an open area of research for over two decades [18], with many different approaches explored with their own advantages and disadvantages but most modern applications still do not simulate fracture in real-time due to the increased complexity of the physics simulation, as well as not allowing the use of pre-computed global illumination since the changing geometry would invalidate the pre-computed lighting values.

Many approaches have been found to model deformation and fracture, typically using pre-calculated models or simplified physics to allow the algorithms to run in real-time and aiming for permissible physics simulation rather than precise physical accuracy.

2.1.1 Voronoi Fracture

One of the earliest developed techniques for modelling fracture in real-time was using voronoi-methods to model brittle fracture. In voronoi fracture, a voronoi tessellation is used to produce a fracture mesh by splitting the base mesh along the lines of a three dimensional voronoi tessellation and using the lines of this tessellation to split the base mesh into multiple new meshes[20].

A major advantage of this method is that the fracture mesh may be precomputed to reduce the runtime requirements of the application but unfortunately this means

that there is limited flexibility in the results that may be obtained and this method has only been used to model brittle fracture.

Recently, Müller et al have developed a variation on this method that allows for far greater realism while still maintaining real-time speeds[14]. Their approach decomposes 3D objects into compounds of convex shapes based on the the voronoi decomposition of the space enclosed by the objects that they call a volumetric approximate decomposition. Unlike a typical voronoi tessellation, however, the voronoi nodes in this approach may be placed manually by the user rather than being drawn from the base mesh, allowing far greater control over the simulation.

In their approach, fracture patterns are predefined but fracture geometry is generated at runtime by aligning the fracture pattern with the impact location and creating new geometry based on the intersection of the fracture pattern with the volumetric approximate decomposition. This provides far greater visual fidelity and realism than naive voronoi methods while also using preprocessing to greatly simplify the algorithm.

2.1.2 Mass-Spring Systems

A more flexible approach is using mass-spring methods to model deformation and fracture. Mass-spring systems are a popular technique for soft-body simulation due to their relative simplicity and reconfigurability [25]. In these systems, objects are modelled as a collection of point masses interconnected with springs in a three dimensional mesh that tend to pull the masses back toward their equilibrium positions. Mass-spring systems may be difficult to implement in stable manner due to the chaotic interactions of the different springs but when utilised effectively, they offer a good approximation of the real-world physics of deformable bodies at real-time speeds.

These systems may be extended to include fracture by placing restrictions on the lengths of the springs connecting the point masses [24] and removing from the system those springs that exceed this length.

Recent work by Levine et al. has also shown that concepts from the area of peridynamics may be used to decouple the spring length from the stiffness [12], allowing for fracture simulation based on stiffness rather than actual spring length. The ability to have variable spring lengths allows for far richer deformations than would otherwise be possible, even allowing for somewhat heterogeneous interior structures to simulated

bodies.

2.1.3 Finite Element Method

Currently, the most physically accurate method for modelling deformation and fracture is the finite element method. In the finite element method, a volumetric tetrahedral mesh is formed for all simulation objects and the equations used to model these individual tetrahedra are used to construct a system of non-linear equations representing the dynamics of the full object. Solving this system of non-linear equations represents a major performance bottleneck and so most real-time implementations use a simplified version of this method to avoid this step.

The earliest implementations of this method used only linear strain approximations in the finite element analysis making it far less computationally expensive but this simplification also introduces volumetric discontinuity in larger deformations[6]. Müller et al. [15] describe a method to use a warped stiffness matrix to allow results comparable to higher-order simulation with only linear computations, allowing for stable deformations in real-time. In this method, a rotation matrix is calculated for each finite element and the forces acting on each element are computed in this rotated frame. In this rotated frame, the deformation of each element is composed of only translation and scaling components removing the non-linear deformations, the element is then rotated back to the original frame in its deformed state, as shown in Figure 2.1.

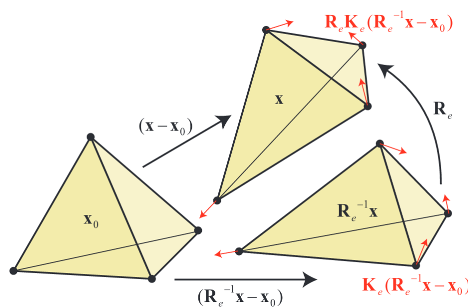

Figure 2.1: Using Warped Stiffness to Approximate Full FEM Analysis[15].

Their later work extended this approach to also deal with fracture by calculating a stress tensor for each element each frame and if the maximum eigenvalue of this exceeds

the material stress tensor, the element is fractured. Elements marked for fracture are then fractured at one of their vertices, the vertex is chosen either at random, or, if one of the vertices is marked as a crack tip, the fracture occurs at this vertex to simulate natural crack propagation through the material. A fracture normal is then calculated as the eigenvector corresponding to the largest eigenvalue of the stress tensor and the vertex is split into two vertices, one on either side of the fracture plane defined by the vertex-normal pair. For all adjacent tetrahedra, the original vertex is replaced by one of the two new vertices depending on which side of the fracture plane the element centre is [16].

Though it has seen some use in modern video games[19], the finite element method is generally thought to be too computationally expensive for use in real-time applications.

Chen et al. also developed a method to perform a coarse grained fracture simulation using the finite element method that is then adaptively refined based on a material strength field to produce high resolution output meshes[5]. This algorithm does not currently work in real-time but may represent a future direction for real-time algorithms.

2.1.4 Position Based Dynamics

All of the previously discussed approaches use force integration where forces acting on an object are accumulated and used to calculate velocities and positions of objects or particles at each time step. An alternative to traditional physics is to use position based dynamics. In position based dynamics, objects are modelled as a system of particles (or vertex positions), whose positions are manipulated directly, with velocities gained implicitly from changes to particle positions. The result is a greatly simplified physics model that aims to be faster than alternatives while still maintaining physics that is realistic enough to be permissible in a real-time context.

Position based dynamics functions by defining a number of constraints on the positions of the particles. Each time step then consists of an explicit Euler integration step that computes new velocities based on applied external forces and new particle positions based on these velocities. Constraints are then generated based on these updated positions and an iterative solver projects these constraints onto the newly calculated positions. Finally, the particles are assigned the constrained particle positions

and their velocities are calculated as $(P_t - P_{t-1})/\Delta t$. The constraints can be used to model a variety of interactions as desired by the developer, for example positions may be constrained based on collisions with other objects in the simulation, or a maximum level of deformation may be enforced through strain constraints. Since the integration step of these methods typically acts independently on all of the particles, they are thought to offer a lot of potential for acceleration through parallelisation.

Shape matching is a form of position based dynamics in which at each time step, the original shape is matched to the deformed configuration, and each particle is translated linearly towards its rest position in this deformed frame, as shown in Figure 2.2.

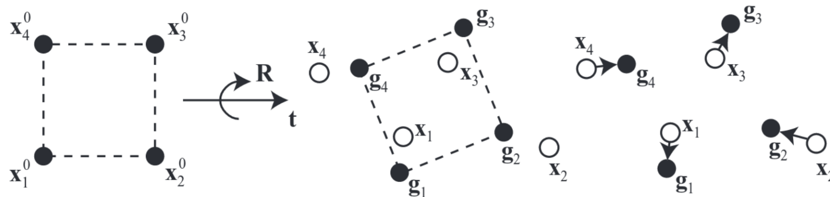

Figure 2.2: Basic Shape-Matching Algorithm[17].

The translation vectors used to transform the original shape into the deformed configuration are simply the centre of mass of the original and deformed shapes, $q_i = x_i^0 - x_{cm}^0$ and $p_i = x_i - x_{cm}$. The calculation of the rotational vector is somewhat more involved, first the rest positions and the deformed positions are computed relative to the object centre of mass in both states, and the matrix A is computed as in Equation 2.1

$$A = \left(\Sigma_i m_i p_i q_i^T\right) \left(\Sigma_i m_i q_i q_i^T\right)^{-1} = A_{pq} A_{qq} \tag{2.1}$$

A_{qq} is symmetric and so contains only scaling and no rotation, therefore the rotation matrix R is the rotational part of A_{pq} which is extracted by the polar decomposition according to Equation 2.2.

$$\begin{aligned} A_{pq} &= RS \\ S &= \sqrt{A_{pq}^T A_{pq}} \\ R &= A_{pq} S^{-1} \end{aligned} \tag{2.2}$$

A stiffness parameter limits the amount that particles may be translated back to their rest state each frame allowing for various materials to be simulated[17]. This approach is relatively simple to implement, very efficient, and unconditionally stable.

It has been shown that this approach may also be used to model plastic deformation and a prototype fracture implementation was shown to be possible in Rivers et al.'s Lattice Shape Matching framework[21].

Recently Jones et al.[11] have developed a clustered shape-matching framework that supports both plastic and elastic deformations, and real-time ductile fracture.

2.1.5 Clustered Shape Matching For Ductile Fracture

Clustered Shape Matching For Ductile Fracture by Jones et. al [11] adds ductile fracture to their real-time shape matching framework, this framework builds upon the seminal paper in Shape Matching by Müller et al.[17]. Previous papers by this group have implemented various other features in this framework, such as a strain limiting iteration [3], clustering, and collision detection[10].

In this dissertation, I am building upon the work of Jones et al. and so a description of their algorithm is given here for reference. This algorithm consists broadly of a plasticity step, followed by shape matching, strain limiting, and finally fracture. Algorithm 1 shows a pseudo-code summary of this method.

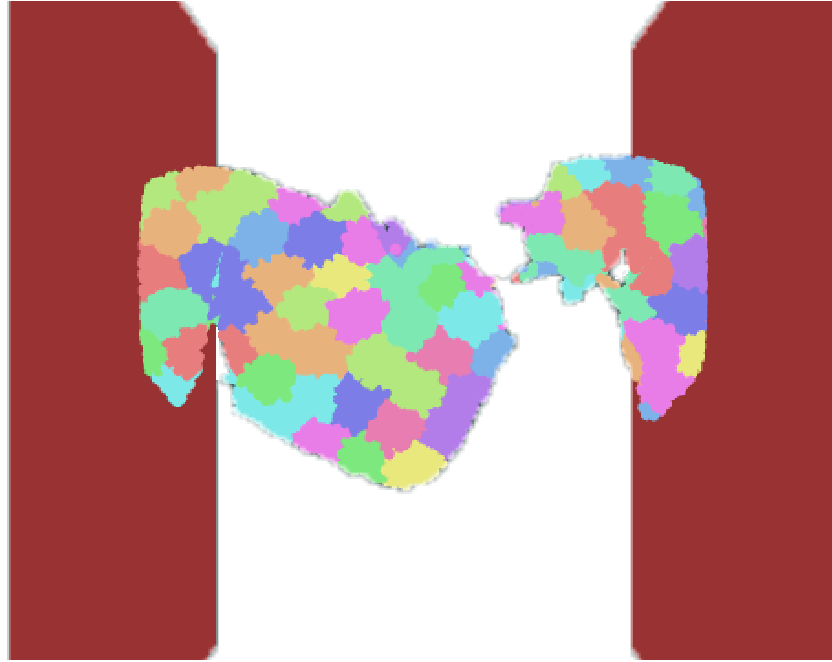

Figure 2.3: A Heart Model is Stretched and Torn in This Framework.

Algorithm 1 Physics time step From Jones et al.[11]

```
1:  for all c ∈ clusters do                                   ▷ Plasticity
2:      c.updatePlasticity()
3:      if c.shouldFracture() then
4:          PotentialFractures.push(c)
5:      end if
6:      for all p ∈ c.particles do                            ▷ Shape Matching
7:          p.incrementGoalPosition()
8:          p.incrementGoalVelocity()
9:      end for
10: end for
11: for no. of constraint iterations do                       ▷ Strain Limiting
12:     strainLimitingIteration(clusters)
13: end for
14: for all pf ∈ PotentialFractures do                        ▷ Fracture
15:     splitCluster(c)
16: end for
17: splitOutlierParticles()                                   ▷ Cleanup
18: removeSmallClusters(); removeLonelyParticles()
```

In this method, clusters are created using an approach similar to fuzzy c-means[7][4]. In the first stage, a k-means iteration is performed, iteratively updating cluster membership by adding particles to the nearest cluster and then updating cluster centres based on all of the particles in the cluster until cluster membership is no longer changing. An additional two-step algorithm is then performed, which also takes into account cluster weights. In this iteration, all particles within a particular radius of a cluster centre are added, weights are assigned based on the distance from the cluster centre, and the cluster centre is recomputed based on the weighted particle positions. The algorithm converges when cluster membership remains constant across iterations. This approach naturally creates overlapping clusters, a necessity in this algorithm to keep objects from falling apart into their constituent clusters.

Dynamics is handled primarily by a standard shape matching algorithm but with an additional strain-limiting iteration step which enforces per-cluster limitations on strain every time step, which ensures that the forces simulated by the algorithm remain bounded and that the simulation remains stable. This approach effectively mitigates the non-linearities that may occur due to effects of multiple overlapping clusters at a point.

Fracture is handled by testing each frame whether a cluster has been stretched beyond its elastic limit and if so a fracture normal is computed perpendicular to the deformation, and the cluster is then split into two separate clusters along this fracture normal. At this point the program will iterate over the particles of the cluster and assign them to one of the new clusters based on their current position relative to this fracture normal. Finally, those particles that belong to clusters on both sides of the fracture plane are duplicated and their mass is split between the two clusters on either side of the plane. This functions similarly to Müller et al.'s Finite Element based fracture described in Section 2.1.3, but uses the A_{pq} matrix from the shape-matching algorithm as an approximation of the strain tensor for a cluster.

In this dissertation, I have parallelised the physics time step summarised in Algorithm 1, while cluster pre-processing still uses the methods developed by Jones et al.

2.2 General Purpose GPU Programming

The two main APIs for GPGPU programming are OpenCL and CUDA, both of which are capable of running highly parallel code on GPUs but since CUDA is a proprietary Nvidia technology, it will only run on Nvidia GPUs. OpenCL however, is an open standard defined by the Khronos group designed for heterogeneous computing meaning that OpenCL code can be run on various heterogeneous computing devices such as CPUs, GPUs, or FPGAs. For this reason OpenCL was chosen for this project as it would allow maximum interoperability and the ability to evaluate the parallel algorithm on both GPU and CPU hardware.

Since OpenCL is a standard defined by the Khronos group implementations differ between manufacturers and operating systems, as well as having different conformance levels on different devices, with the latest offerings from AMD and Intel supporting OpenCL 2.0 while all Nvidia deices and many others on the market today support only the OpenCL 1.2 standard[1]. In order to support the vast majority of platforms and to target multiple test machines, the target OpenCL version of this project was set at 1.2 but further work may explore the use of OpenCL 2.0 features, such as dynamic parallelism and shared virtual memory. At this time fully supporting all of the major desktop hardware vendors (AMD, Intel, and Nvidia), would necessitate writing parallel code in OpenCL 1.2, OpenCL 2.0, and CUDA, switching code path depending on the platform.

2.3 OpenCL Architecture

The OpenCL platform model consists of a host connected to one or more compute devices. The compute devices contain a number of compute units, which are further broken down into many processing elements. Typically the host is a CPU program, and the compute devices may be CPUs, GPUs, or FPGAs.

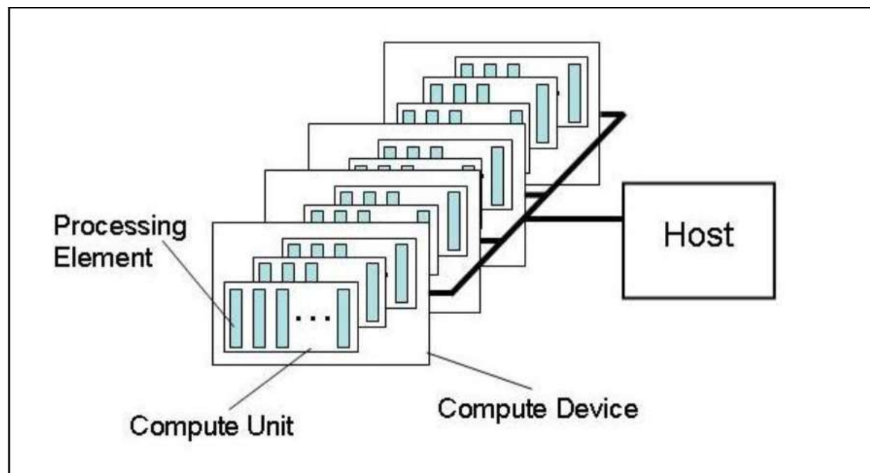

Figure 2.4: OpenCL Platform Model[2].

The processing elements within a compute unit will execute either a single stream of instructions as SPMD (single program, multiple data), or execute in lockstep as SIMD (single instruction, multiple data), so that both data parallel and task parallel programming models are available. The most common usage, and that employed by this project is to use SPMD to run the same program on different data in parallel. In this model, programs are written that will execute over a range of work-items and these programs are called OpenCL kernels.

All memory to be used by the OpenCL kernels must be loaded into buffers to be used by the compute device and sharing data between host and device memory is typically a very costly operation. A notable exception is that when using a CPU as a compute device, the OpenCL memory is allocated on the main system memory and so a shared pointer may be used between compute device and host.

The OpenCL memory model consists of four memory regions, each serving a different purpose.

Global memory allows read/write access to all work-items running on a device. Constant memory is an area of memory that does not change during the execution of a kernel, constant memory may only be written to by the host and is typically implemented as a subset of the global memory area.

Local memory is fully accessible by all work-items in a work-group. Local memory may be implemented on device as either a dedicated memory space or a partitioned

area of global memory depending on the vendor's implementation.

Private memory is an area of memory visible only to an individual work-item and is typically used for variables defined within a kernel. An overview of this model is shown in Figure 2.5.

Of these memory areas, global memory is the largest, slowest memory area, while local and private memory areas are typically allocated in caches or registers depending on the implementation and the amount of memory space needed .

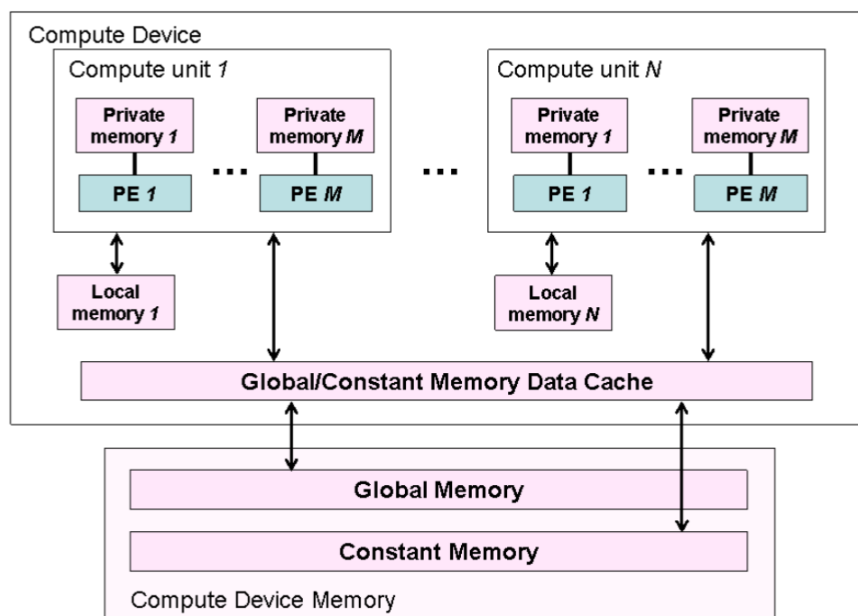

Figure 2.5: OpenCL Memory Model [2].

In OpenCL, the state of memory visible to each work-item is not guaranteed to be consistent across the collection of work-items at all times. This means that reading from memory may not provide up-to-date data and if many work-items need read/write access to the same buffer, synchronising threads to ensure memory consistency will represent a considerable cost to performance.

Atomics for integer operations are also supported for lock-free programming but floating point atomics are not supported because it is not guaranteed that they will be supported natively by all devices.

Chapter 3

Experimental Design

This chapter details the measurements that were made to evaluate the implementation as well as a brief overview of the devices it was tested on.

3.1 Target Platforms

All of the experiments were performed on two different work-stations to ensure portability of the implementations and to observe any significant requirements for ideal performance. A summary of the hardware in these different work-stations is summarised in Table 3.1. All of the data in this table were gathered directly from the hardware through the OpenCL API.

	Test Machine 1		Test Machine 2	
Device Type	CPU	GPU	CPU	GPU
Vendor	Intel	Nvidia	Intel	AMD
Device	Xeon E3-1240	Quadro K2000	Core i5 - 4670K	Radeon R9 390
Compute Units	8	2	4	40
Clock Speed	3400 Mhz	954 Mhz	3400 Mhz	1040 Mhz
Global Memory Size	17 GB	2 GB	24 GB	8 GB
Global Memory Cache	256 KB	32 KB	256 KB	16 KB
OpenCL Version	1.2	1.2	1.2	2.0

Figure 3.1: System Information of Test Platforms as Reported by the OpenCL API.

The various architectures of the devices on these two platforms allow for testing

in environments with different limitations to determine the ideal configuration, as well as any shortcomings that increase the hardware requirements of the implementation. Though the global memory space available in these devices is highly variable, the minimum value of 2GB is still more than enough to hold all of the simulation data in memory so this limitation is not likely to be relevant. Of interest however is that the GPU memory is typically substantially faster than the system RAM available to the CPU. The CPUs, however have a far larger cache area devoted to OpenCL global memory, so they will likely deal considerably better with memory intensive sections of code.

Unfortunately, due to the heterogeneous nature of OpenCL, the reported values of compute units has a different meaning across the different devices. For instance, the 8 compute units on the Xeon, compared to the Core i5's 4 is due to the former supporting hyperthreading, meaning that there are the same number of physical CPU cores but the Xeon may achieve greater parallelism in some cases by utilising hardware that would otherwise be idle [13], which will not give as significant a performance increase as having double the physical cores on the device. it is also worth noting that due to the inclusion of SSE vector extensions, these processors are also capable of 4-way SIMD parallelism allowing far greater throughput on parallel data sets.

Similarly to the CPUs, the GPUs have different definitions of a compute unit with the equivalent in Nvidia terminology being a Streaming Multiprocessor. In the case of these two GPUs, the Nvidia chip reports two compute units that map to 384 CUDA cores while AMD reports 40 compute units, that correspond to 2560 stream processors. More generally, this means that both of the GPUs have many more SIMD cores per compute unit allowing for far greater levels of parallelism than one would expect simply from Table 3.1.

3.2 Measurements

This section details the measurements taken to evaluate the implementation and the methodology used to gather the data. Platform specific data was gathered using Intel's VTune Amplifier, Nvidia's NSight, and AMD's CodeXL. Unfortunately a standard tool to gather this data could not be found so the values reported by these tools are treated as comparable where possible but some discrepancy is assumed.

All of the measurements were taken running the broken heart demo as it had the highest number of particles and clusters of the available demos, as well as having the most fractures meaning that it represented the best stress-test available. The figures used here are the average of five executions of the simulation in order to ensure that these measurements represent the average case and not outlier behaviour.

3.2.1 Frame Time

The main measurement in these experiments is frame time, the amount of time to simulate and render one frame. No external tools were used for this measurement, the C++ standard library utility chrono was used for time measurements, which were then written out to a CSV file upon program exit.

The main performance analysis was performed with both rendering and physics enabled to measure the speed of the full implementation as it would be used in a real world application.

3.2.2 Hot Spots

The hot spots in the code where most of the time is being spent were measured for the Intel and Nvidia devices to determine where the main bottlenecks in the code were but unfortunately a bug in the AMD profiling software caused it to crash when attempting to gather the same data.

3.2.3 Memory Analysis

A detailed memory analysis was carried out for the Intel and AMD devices, however Nvidia's Nsight did not allow such fine-grained analysis for OpenCL and so no data was gathered for this device.

The level of granularity of the results is also highly variable between the AMD and Intel devices so the potential for direct comparisons was limited but where possible, values recorded on each were compared directly.

Chapter 4

Implementation

This section details the main steps required to implement the multi-threaded implementation. A lot of further development effort was spent tackling platform specific bugs and ensuring portability across the various devices tested.

4.1 OpenGL Renderer

The implementation by Jones et. al [11] used an open source rendering engine called Ogre3D to handle the rendering of the particles. This caused a considerably increased API overhead and provided only partial support for modern OpenGL functionality. To alleviate this rendering bottleneck and allow for OpenGL and OpenCL interoperability, a custom renderer was written using OpenGL 3.3 that uses instanced rendering to efficiently render large numbers of particles each frame. This updated renderer reduced frame times by approximately 90%.

At the time of writing, the rendering engine is not at feature parity with the Ogre3D engine. Though fast rendering allows better analyisis of the physics implementation, more complex rendering effects may be added in a later revision.

4.2 OpenCL Optimisation

Though often considerably faster, a naive GPU implementation of an algorithm using OpenCL is not guaranteed to be significantly faster. OpenCL kernels are often memory-

bound and so peak performance will be achieved when the ratio of computation to memory accesses is large. Also of importance is the memory hierarchy (discussed in Section 2.2), making good use of the faster private and local memory areas can significantly reduce the cost of memory accesses, operations bound for global memory should be kept to an absolute minimum.

A GPU is built for maximum bandwidth and so it is adept at performing a lot of computations very quickly but execution cores lack the branch prediction hardware that is ubiquitous on CPUs. In the case of branching execution paths, a GPU will typically compute the result of both paths and discard whichever is not needed. For this reason, it is important to keep branching to an absolute minimum when designing algorithms for use on a GPU so where possible branching code was removed and if possible, the branching was handled by the host device deciding whether or not to execute a kernels.

A considerable performance bottleneck of any parallel algorithm is thread synchronisation. The simplest form of synchronisation is to split the algorithm into multiple kernels so that the state of the global memory space is known to be consistent when each kernel is dispatched, this means that the synchronisation is handled by the host, as it will only dispatch a new kernel to the compute device once the previous kernel has completed processing, effectively limiting the level of parallel processing possible in order to avoid data races. Another approach is to implement mutexes in the kernel code or to use memory barriers that halt execution of individual work-items until all work items have reached the barrier command but both of these approaches have shown poor scaling with increasing number of work-items[8]. For this reason, this implementation is entirely lock-free, with the host device handling most of the synchronisation and atomic operations used for the few times that memory synchronisation was necessary within a kernel.

4.3 Parallel Physics

The original single-threaded implementation consists of many steps that iterate over either all particles or all clusters in the simulation. The first step in implementing a parallel implementation was to write an OpenCL kernel for each of these iterations that would instead run the same code in parallel for each cluster or particle. Much of

the code base worked reasonably well once implemented in this naive fashion but there were a number of issues with these preliminary kernels.

The first major issue with this initial implementation was those areas where clusters iterate over and update all of their particles. In the single-threaded case there is no issue but in a multi-threaded environment, this will create a data-race as multiple clusters attempt to update the same particles. For this reason the particle object was updated so that each particle was aware of its weight value in each of the clusters to which it belongs so that the kernel may instead run on every particle so that the work-items will have exclusive access to the particle they are updating. Each particle can then read values from each of its clusters to update its properties without causing a data race. This allows for far greater parallelism and avoids the issue of having to synchronise memory accesses across many work-items. Similarly updating the particle goals requires the calculation of a matrix $T = R\,(c - R\hat{c})$, where c and $\hat{c}$ are the current and original centres of mass of the cluster, and R is the rotation matrix described in Section 2.1.4. In the parallel implementation, instead of calculating this matrix and incrementing the goal positions of each particle in the cluster directly, the matrix is simply stored in memory and accessed by the subsequent particle kernel that performs the update for all particles in parallel. It was found that the increased parallelism and lock-free nature of this approach more than offset the cost of the extra memory operations to store the matrix and re-read it from memory.

Areas where the clusters require read-only access to the particles in order to update their own properties were left unchanged as this interaction was already thread safe and it was found that the increased parallelism achieved by running kernels on all particles was completely mitigated by the time threads spent waiting for exclusive write-access to each of their clusters.

Where possible, kernels that operated on the same data were then merged to avoid the API overhead in dispatching a new kernel, and to avoid unnecessarily returning control to the host device as much as possible. Commonly used data was also loaded into kernel private memory at the beginning of each kernel to make use of the faster private memory area rather than performing more expensive accesses to the slower global memory area.

The fracturing algorithm, (summarised in Algorithm 2) unfortunately affords less potential for parallelism since it largely operates only on a subset of the full set of

Algorithm 2 Fracturing Algorithm From Jones et al.[11]

```
1: for all f ∈ potentialFractures do
2:     if clustershouldfracture then
3:         f.cluster.split()
4:         for all p ∈ c.particles do
5:             for all pc ∈ p.clusters do
6:                 if dot(p.pos, f.norm) > 0 ≠ dot(pc.worldCom, f.norm) > 0 then
7:                     splitParticle(p)
8:                 end if
9:             end for
10:        end for
11:    end if
12: end for
13: removeSmallClusters()
14: removeLonelyParticles()
```

clusters and particles. The first step runs in parallel only for each potential fracture, meaning that the absolute upper limit on the number of parallel threads is equal to the number of clusters in the simulation, but in practice this is much smaller and in this implementation was limited to 64 .

The single threaded implementation then iterates over all of the particles in the affected cluster, and checks all of the clusters this particle belongs to and if any of these are not on the same side of the fracture as the particle, the particle is split into two particles, one in each cluster on either side of the fracture plane. For the parallel algorithm, this step is instead completed as two separate kernels, with the fracturing cluster simply marking all of its clusters as potentially splitting and in a subsequent step the particles all run in parallel, testing all of the fracture normals against all of their clusters. This affords greater parallelism since the algorithm can now run in parallel on each individual particle with the caveat that it is still iterating over all of its clusters for each fracture in this simulation step but since the number of fractures and the number of clusters per particle is relatively small, this was not seen to be a significant performance bottleneck. Unfortunately, this approach requires a significant amount of branching and so is not ideal for GPU devices.

The subsequent steps all operate on the entire set of clusters or particles but since each of these alters the number of clusters or particles, they do require intermediate

steps to ensure synchronisation between the particles and clusters. For example, after removing small clusters, an additional step is required in which the particles' cluster arrays are cleared and each cluster then atomically adds itself to the cluster array of each of its particles to ensure that none of the particles are affected by clusters that no longer exist. A similar step is also run after particles are added or removed from the simulation. These synchronisation steps are composed of almost entirely memory operations and so are likely to be a significant performance bottleneck but unfortunately no way was found to eliminate them entirely.

The changing number of particles and clusters in the fracturing algorithm is also problematic because, though attempts have been made to implement a dynamic memory system in OpenCL [23], the OpenCL buffers themselves are of fixed size. For this reason, bit arrays were implemented that would indicate to the OpenCL device which particles and clusters are active in the simulation. Therefore, to delete from a buffer requires only a single AND operation to clear the relevant bit in the bit array giving $O(1)$ complexity, and insertion into the buffer requires iteration over the bit array until the first zero index is found, resulting in worst-case $O(n)$ complexity (excluding the cost of the write operation). This approach requires prior knowledge of what the maximum size of the buffers should be, which will vary depending on the simulation objects and the interactions modelled. In this implementation the buffer sizes were set as 1.5 times the number of clusters and particles present at the beginning of the simulation and this has been found to avoid buffer overflow but it is possible that these buffers may be tuned better to minimise memory use.

4.4 Shortcomings

Due to time constraints collision detection and response was not implemented and so certain simulations that were supported in the original implementation are not supported in the OpenCL parallel implementation. This is unfortunate because the algorithm lends itself well to fast and accurate collision detection due to the natural space partitioning of the clusters and particles that make up the simulation objects.

It is also worth noting that though OpenCL is designed as a heterogeneous computing framework, methods often exhibited different behaviour across different platforms leading to considerable issues ensuring portability of code, including a major re-write

when adding support for CPU devices because some of the C++ wrapper API methods used incompatible default values, and so large amounts of initialisation and kernel execution code had to be implemented instead with different C methods. Even with this fallback however, there are still many points where the code diverges and uses a different path depending on whether the compute device is a CPU or GPU, and even different methods for loading data into buffers depending on whether the target GPU is an AMD or Nvidia device. Future work may explore the potential of newer technologies like OpenGL 4+ compute shaders or the Vulkan API to allow true portability, at least between different GPU devices.

Chapter 5

Results

Jones et al. include many sample scenes to be run with their application, ranging in complexity from simple deformations to fractures caused by multiple projectiles. Due to time constraints not all have been tested to work with this new implementation but those that have are listed in Table 5. The total dataset from which the results in this chapter were drawn is available in Appendix 1 for reference.

Example	# Particles	# Clusters	CPU - ST	CPU - MT	GPU
Broken Heart	20132	100	31.82	17.6	28.17
Twisting Bar	5317	40	10.67	6.05	42.96
Twisting Bar Fracture	5317	40	11.15	6.63	42.81

Figure 5.1: Summary of Test Scenes Showing Average Frame Times For Different Configuration of This Application (measured on Core i5 and R9 390 devices).

5.1 Frame Times

This section explores the differences between the different configurations in the time taken to simulate and render each frame. This is the primary metric used to evaluate the different configuations of this project.

5.1.1 Test Machine 1

On the first test machine, the multi-threaded GPU implementation was the slowest configuration to complete the physics calculations, but is the fastest to complete the rendering step.

This is a surprising result since the far larger number of cores on the Nvidia GPU should allow it to perform substantially better in highly parallel applications. It would seem from these results that core count is not the primary bottleneck of the algorithm as the Nvidia device would show far better peformance if that were the case.

When the simulation is running on the GPU, the entire particle buffer remains resident in the GPU memory, avoiding costly data transfers and leading to rendering times that are approximately half of the other two configurations. As can be seen from Figure 5.3, however, this rendering efficiency has an insignificant effect on the total fame times.

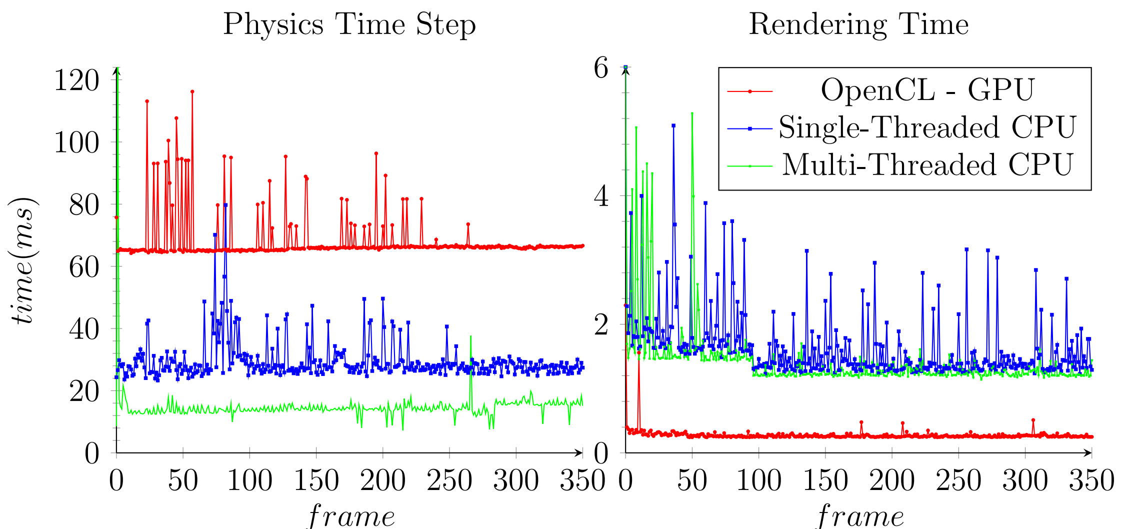

Figure 5.2: Physics and Rendering Performance in Various Configurations on Test Machine 1.

On the CPU side, the multi-threaded implementation showed an average of 40% improvement over the original single-threaded implementation due to increased CPU utilisation improving processing throughput in the physics time step. Though a significant performance increase, this shows a less than linear improvement with increasing core count and suggests there may be low core utilisation in the multi-threaded implementation.

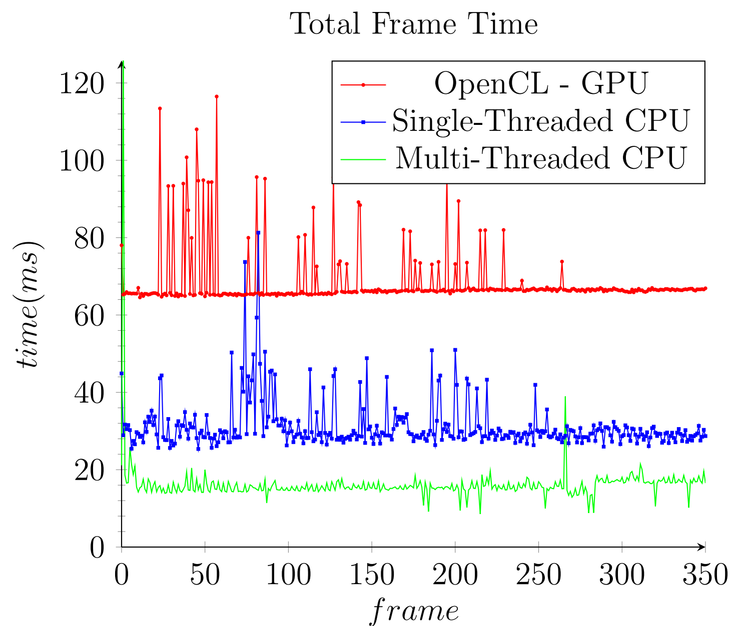

Figure 5.3: Relative performance of different implementations on Test Machine 1.

Also of note is that the frame-times in the case of the multi-threaded CPU are far more stable, not suffering from the substantial increases in frame times that the GPU and the single-threaded CPU implementations exhibit as soon as the object begins fracturing. This is arguably more important than the actual reduction in frame times as it shows more of an ability to scale with increasing complexity of the simulation.

5.1.2 Test Machine 2

On the second test machine the GPU executed the physics time step in slightly lower time than the single threaded CPU implementation and showed a substantial improvement in rendering times resulting an average improvement of 9.9% over the single-threaded CPU implementation. However, once again the multi-threaded CPU was shown to be substantially faster despite the slower rendering times.

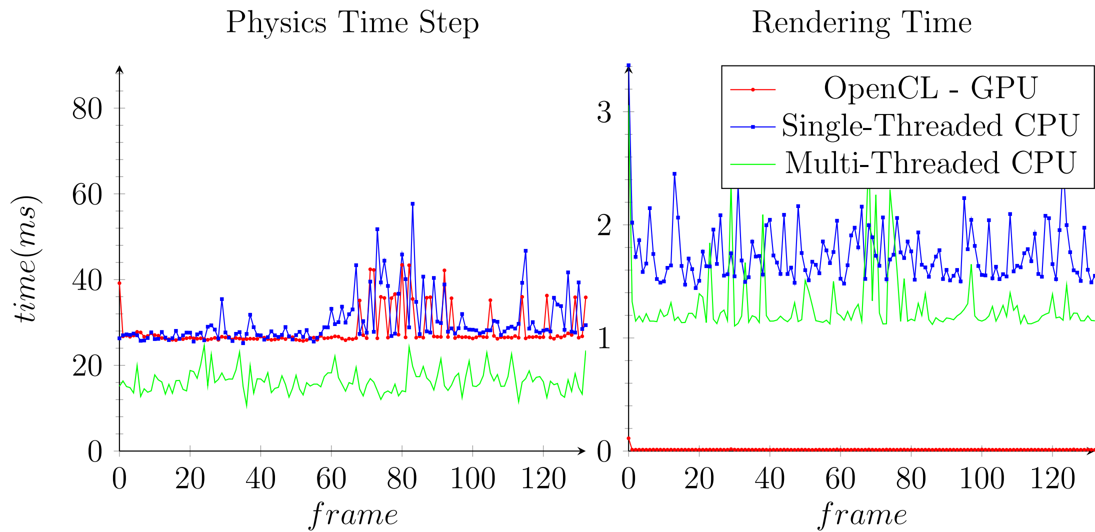

Figure 5.4: Physics and Rendering Performance in Various Configurations on Test Machine 2.

The multi-threaded CPU performed almost identically as on the first test machine, showing that the impact of hyper threading on the overall speed was negligible for this algorithm.

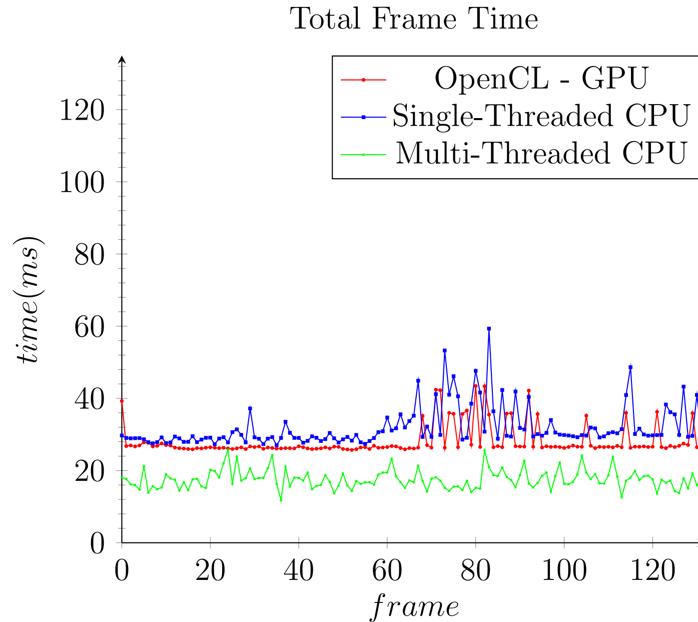

Figure 5.5: Relative performance of different implementations on Test Machine 2.

As expected, the R9 390 performs considerably better than the Quadro K2000 but as Figure 5.5 shows, the GPU implementation still under performs, showing clearly that the algorithm cannot scale effectively with increasing core-count.

5.2 Hotspot and Memory Analysis

Figure 5.6 shows the miss rates in the last level cache for the Intel CPUs and the AMD GPU, as mentioned in Section 3.2.3 the same data was not available for the Nvidia GPU and so it was omitted from this plot. It is expected that the Intel CPUs would perform better in these circumstances due to the vastly larger cache area allocated to OpenCL global memory on the Intel devices but the difference shown is significantly more substantial than could be expected.

This is likely to do with the fact that, on a hardware level the difference in caches between the devices is even more substantial than the 16x difference shown in Table 3.1, the R9 390 has only 1 MB of L2 cache shared between all compute units while the Xeon chip and the Core i5 have 8 MB and 6 MB L3 Caches respectively that are shared between all CPU cores, giving a far greater ratio of cache space to execution cores and

allowing them to almost completely avoid loads from the system RAM for the duration of the simulation.

By analysing values for individual kernels it was found that the worst case cache performance was experienced in short kernels that simply read and update a single value, exhibiting an especially poor ratio of memory operations to ALU instructions. Another significant issue is those kernels in which clusters are updated based on values held by all of their particles. This is likely due to the fact that the position of a particle in the buffer is not guaranteed to be related to its spatial positioning leading to a largely random access pattern by the cluster kernels that is likely to cause some thrashing of the small cache on the GPU. This behaviour could be improved by storing particles in a BSP tree or similar spatial partitioning scheme, or by storing all of the particle attributes in separate buffers to improve locality of reference.

Figure 5.7 shows how this difference of cache architecture affects the AMD GPU, showing that the vast majority of time is spent on memory operations while the vector and scalar ALUs perform relatively little work. Clearly this is why the algorithm does not scale effectively with increased core counts since the main bottleneck is memory accesses.

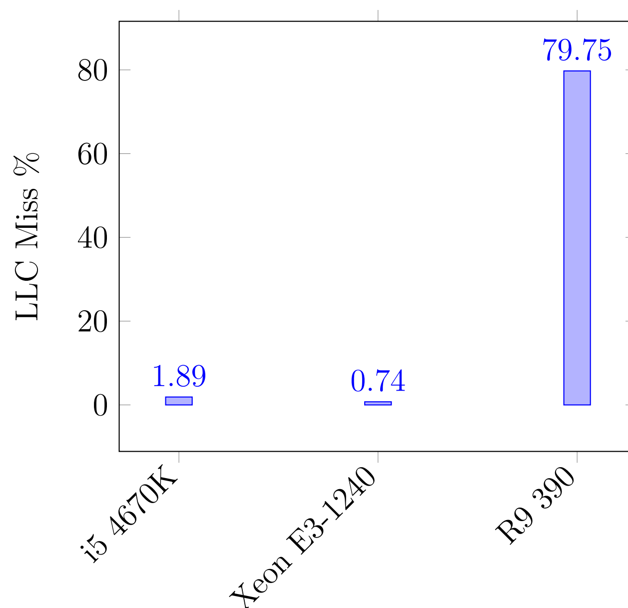

Figure 5.6: Average Percentage of Misses in the Last Level Cache for the Intel and AMD Devices.

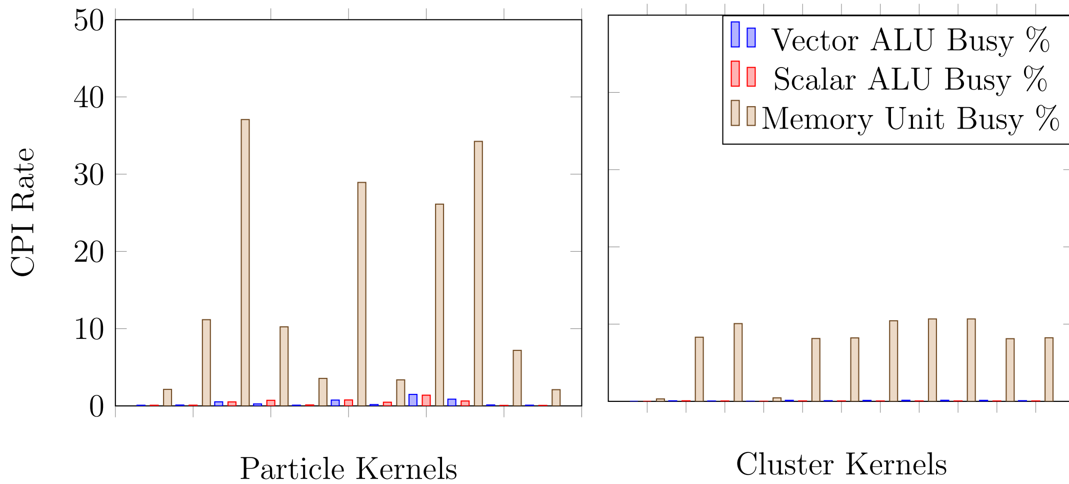

Figure 5.7: Comparison of ALU and Memory Unit Utilisation on R9 390 Device.

Cycles per instruction is a useful metric as it shows how quickly instructions are passing through the CPU pipeline. Ideal values of CPI are less than or equal to one, meaning that at least one instruction is finishing execution in every clock cycle. A low CPI indicates that the hardware in the CPU is being utilised effectively and instructions are not stalled in one pipeline stage for a significant amount of the clock cycle. A high CPI indicates a large amount of stalls due to slower operations relative to the amount of faster operations that are passing through the pipeline. Figure 5.8 shows the CPI values for each kernel in the simulation running on the Core i5 device split up by whether they operate on all clusters or all particles, the percentage of misses in the last level cache is also shown to illustrate that many of the largest bottlenecks are driven by a large proportion of cache misses.

It can be seen from Figure 5.8 that one of the particle kernels suffers from particularly high CPI owing largely to the miss percentage in the last level cache. This illustrates a significant disadvantage to this algorithm as this kernel is used to reset the particle goal positions and velocities for each frame, which are then accumulated based on cluster positions. This is essentially worst-case performance since the kernel consists of almost exclusively memory operations and must be executed before any other kernels in the physics time step.

This illustrates a major issue with the algorithm used. The particle kernels typically require access to the cluster buffer and vice-versa, meaning the data in these buffers

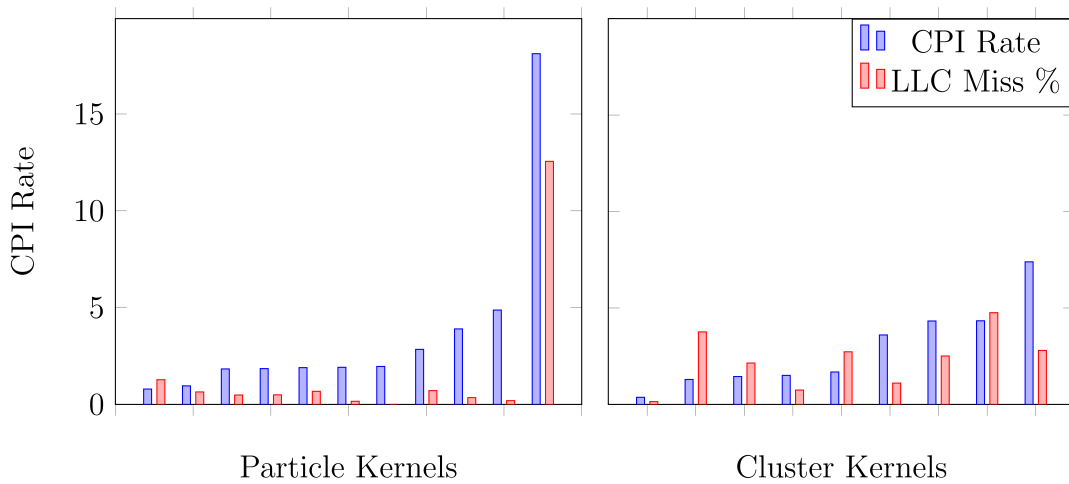

Figure 5.8: CPI and LLC Miss % For All Kernels on the Core i5 Device.

must be updated many times throughout each time step. Of the top four cluster kernels by CPI value, three of them are simply updating cluster values that the particles will be accessing in subsequent kernels. This requirement for synchronising data over the course of a single time step is a considerable performance bottleneck and a significant challenge to effectively implementing parallelised clustering.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The parallel implementation of clustered-shape matching presented here achieves best-case performance increases of approximately 40% but due to memory bottlenecks, fails to scale on GPU devices and exhibits generally poor cycles per instruction on multicore CPU devices.

As can be seen in Section 5.2, the requirement to keep properties in sync between particles and clusters results in many areas where there is a significant memory bottleneck limiting the performance increase that may be gained from parallelism. The CPUs tested managed to mitigate this bottleneck somewhat through their deeper cache hierarchy and large L3 cache shared amongst all cores but the overall performance is still significantly limited. Therefore, for this particular algorithm a multi-core CPU performs far better than a many-core GPU but it may be possible to implement the same rich simulation of deformation and fracture using position-based-dynamics while also avoiding this memory bottleneck, by using spatial partitioning to increase the cache efficiency of the algorithm.

In Figure 5.4 a non-negligible reduction in rendering time is observed from keeping all of the data resident in GPU memory for the duration of the simulation, allowing data transfers across the PCIe bus to be avoided. Maintaining this increased performance is further motivation for a parallel implementation that is not memory bound, and may be executed effectively on GPU devices.

Particle based dynamics then does offer a great potential for parallelisation but this clustered approach has considerable limitations in the performance gain that may be archived due to the significant memory overhead necessary to keep data consistent across particles and clusters.

6.2 Future Work

As mentioned in Section 2.2, improved performance may be achieved by implementing different code paths depending on the hardware present, especially advantageous would be allowing the use of OpenCL 2.0 on AMD GPUs and CUDA version 5+ for Nvidia GPUS. Better performance may be achieved by using a different framework depending on the platform, while also allowing a measurement of the potential performance improvement from these more modern APIs. An aternative method would be to use OpenGL 4 compute shaders or the new Vulkan API, perhaps performing a comparison to determine the relative performance of the different APIs.

Adding parallelised collision detection, such as the implementation of Greß et al. [9] would allow the parallel implementation to simulate more complex scenes, and bring the parallel implementation to feature parity with the single-threaded implementation.

Improved rendering, and in particular, an efficient method for particle skinning could greatly improve the appearance of the simulation, and add to its viability for use in real-time applications.

Appendix1 - Detailed Results

This section shows the full dataset from which the results were drawn. Again all figures were recorded executing the broken heart demo program.

Nvidia Quadro K2000 Profiler Data

kernel	Device Time (s)	min(s)	avg(s)	max(s)
TestOutsideTiltingPlane	0	0	0	0
TestOutsideTwistingPlane	0	0	0	0
ClearParticleGoals	0.26	$2.41 \cdot 10^{-4}$	$2.89 \cdot 10^{-4}$	$3.7 \cdot 10^{-4}$
UpdateClusterPlasticities	3.54	$3.85 \cdot 10^{-3}$	$3.92 \cdot 10^{-3}$	$7.64 \cdot 10^{-3}$
UpdateParticleGoals	0.94	$9.31 \cdot 10^{-4}$	$1.04 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$
ZeroParticleGoalPositions	0.25	$5.48 \cdot 10^{-5}$	$6.88 \cdot 10^{-5}$	$1.23 \cdot 10^{-4}$
StrainLimitingIteration	9.4	$2.55 \cdot 10^{-3}$	$2.61 \cdot 10^{-3}$	$2.68 \cdot 10^{-3}$
StrainLimitingGoalUpdate	3.86	$9.43 \cdot 10^{-4}$	$1.07 \cdot 10^{-3}$	$1.15 \cdot 10^{-3}$
UpdateParticleVelocities	0.13	$1.17 \cdot 10^{-4}$	$1.39 \cdot 10^{-4}$	$1.7 \cdot 10^{-4}$
UpdateClusterComs	1.29	$1.4 \cdot 10^{-3}$	$1.43 \cdot 10^{-3}$	$1.47 \cdot 10^{-3}$
FracturePotentialSplits	$4.53 \cdot 10^{-2}$	$4.88 \cdot 10^{-6}$	$5.03 \cdot 10^{-5}$	$6.46 \cdot 10^{-3}$
SplitParticles	0.81	$4.32 \cdot 10^{-5}$	$9.02 \cdot 10^{-4}$	$6.36 \cdot 10^{-2}$
SplitOutliers	0.62	$6.23 \cdot 10^{-4}$	$6.88 \cdot 10^{-4}$	$8.04 \cdot 10^{-3}$
CullSmallClusters	0.39	$3.98 \cdot 10^{-4}$	$4.37 \cdot 10^{-4}$	$4.55 \cdot 10^{-4}$
RemoveLonelyParticles	$4.49 \cdot 10^{-2}$	$4.18 \cdot 10^{-5}$	$4.98 \cdot 10^{-5}$	$5.66 \cdot 10^{-5}$
ClearParticleClusters	$7.04 \cdot 10^{-2}$	$3.14 \cdot 10^{-5}$	$3.91 \cdot 10^{-5}$	$6.98 \cdot 10^{-5}$
ClearClusterMembers	$9.64 \cdot 10^{-3}$	$6.04 \cdot 10^{-6}$	$1.07 \cdot 10^{-5}$	$1.6 \cdot 10^{-5}$
CountClusters	5.15	$2.82 \cdot 10^{-3}$	$2.86 \cdot 10^{-3}$	$3.02 \cdot 10^{-3}$
CountClustersPerParticle	1.11	$1.03 \cdot 10^{-3}$	$1.23 \cdot 10^{-3}$	$1.35 \cdot 10^{-3}$
UpdateClusterProperties	5.69	$3.1 \cdot 10^{-3}$	$3.16 \cdot 10^{-3}$	$3.27 \cdot 10^{-3}$
UpdateClusterTransforms	4.46	$1.6 \cdot 10^{-3}$	$1.65 \cdot 10^{-3}$	$1.7 \cdot 10^{-3}$
MovingPlaneInteraction	0.21	$1.84 \cdot 10^{-4}$	$2.29 \cdot 10^{-4}$	$3.01 \cdot 10^{-4}$
TwistingPlaneInteraction	0	0	0	0
TiltingPlaneInteraction	0	0	0	0
UpdateFpAndCom	1.13	$1.19 \cdot 10^{-3}$	$1.25 \cdot 10^{-3}$	$1.28 \cdot 10^{-3}$

Core i5 Profiling Data

Hotspots

kernel	Utilisation Time (s)	Instructions Retired	CPI Rate
ClearClusterMembers	0.54	$1.04 \cdot 10^{9}$	1.9
ClearParticleClusters	2.4	$5.87 \cdot 10^{9}$	1.44
ClearParticleGoals	1.18	$2.35 \cdot 10^{9}$	1.83
CountClusters	0.1	$2.22 \cdot 10^{8}$	1.68
CountClustersPerParticle	0.39	$1.75 \cdot 10^{9}$	0.79
CullSmallClusters	0.38	$3.44 \cdot 10^{8}$	3.9
FracturePotentialSplits	3.62	$2.7 \cdot 10^{9}$	4.87
MovingPlaneInteraction	0.54	$2.07 \cdot 10^{9}$	0.96
RemoveLonelyParticles	1.09	$5.41 \cdot 10^{8}$	7.39
SplitOutliers	0.54	$5.47 \cdot 10^{8}$	3.6
SplitParticles	2.2	$6.24 \cdot 10^{9}$	1.29
StrainLimitingGoalUpdate	$2 \cdot 10^{-4}$	$4.76 \cdot 10^{6}$	0.4
StrainLimitingIteration	0.29	$5.28 \cdot 10^{8}$	1.96
UpdateClusterComs	6.26	$1.17 \cdot 10^{10}$	1.92
UpdateClusterPlasticities	2.19	$1.82 \cdot 10^{9}$	4.32
UpdateClusterProperties	1.11	$2.17 \cdot 10^{9}$	1.85
UpdateClusterTransforms	$3 \cdot 10^{-3}$	$5.1 \cdot 10^{7}$	0.37
UpdateFpAndCom	0.35	$4.04 \cdot 10^{8}$	2.84
UpdateParticleGoals	1.65	$3.27 \cdot 10^{8}$	18.11
UpdateParticleVelocities	4.2	$1.01 \cdot 10^{10}$	1.5
ZeroParticleGoalPositions	0.53	$2.21 \cdot 10^{9}$	0.88

Memory

kernel	CPU Time	Loads	Stores	L3 Misses	L3 Miss Percent
ClearParticleClusters	0.13	$1.44 \cdot 10^7$	$2.4 \cdot 10^7$	0	0
ClearParticleGoals	0.72	$3.76 \cdot 10^7$	$6.54 \cdot 10^7$	$4.72 \cdot 10^6$	12.55
CountClusters	1.57	$2.15 \cdot 10^8$	$1.72 \cdot 10^8$	$5.4 \cdot 10^6$	2.51
CountClustersPerParticle	1.51	$4.45 \cdot 10^8$	$3.69 \cdot 10^8$	$2.2 \cdot 10^6$	0.49
CullSmallClusters	0.43	$2.2 \cdot 10^8$	$2.08 \cdot 10^8$	$3.2 \cdot 10^5$	0.15
FracturePotentialSplits	$5.62 \cdot 10^{-2}$	$3.52 \cdot 10^7$	$1.71 \cdot 10^7$	$9.6 \cdot 10^5$	2.73
MovingPlaneInteraction	0.7	$2.97 \cdot 10^8$	$2.01 \cdot 10^8$	$1.44 \cdot 10^6$	0.49
RemoveLonelyParticles	0.6	$2.81 \cdot 10^8$	$1.44 \cdot 10^8$	$2 \cdot 10^6$	0.71
SplitOutliers	0.31	$2.61 \cdot 10^8$	$8.85 \cdot 10^7$	$1.68 \cdot 10^6$	0.64
SplitParticles	0.36	$3.14 \cdot 10^8$	$9.49 \cdot 10^7$	$4 \cdot 10^6$	1.28
StrainLimitingGoalUpdate	3.29	$2.14 \cdot 10^9$	$5.48 \cdot 10^8$	$3.44 \cdot 10^6$	0.16
StrainLimitingIteration	7.01	$3.12 \cdot 10^9$	$7.74 \cdot 10^8$	$2.32 \cdot 10^7$	0.74
UpdateClusterComs	0.5	$2.9 \cdot 10^8$	$1.13 \cdot 10^8$	$3.2 \cdot 10^6$	1.11
UpdateClusterPlasticities	2.08	$4.9 \cdot 10^8$	$9.78 \cdot 10^7$	$2.33 \cdot 10^7$	4.76
UpdateClusterProperties	2.65	$9.39 \cdot 10^8$	$5.92 \cdot 10^8$	$3.53 \cdot 10^7$	3.76
UpdateClusterTransforms	2.49	$1.29 \cdot 10^9$	$6.64 \cdot 10^8$	$2.77 \cdot 10^7$	2.14
UpdateFpAndCom	1.44	$7.03 \cdot 10^8$	$2.12 \cdot 10^8$	$1.97 \cdot 10^7$	2.8
UpdateParticleGoals	2.23	$6.37 \cdot 10^8$	$1.9 \cdot 10^8$	$1.24 \cdot 10^6$	0.19
UpdateParticleVelocities	2.27	$4.22 \cdot 10^8$	$1.54 \cdot 10^8$	$1.48 \cdot 10^6$	0.35
ZeroParticleGoalPositions	0.46	$7.68 \cdot 10^7$	$5.65 \cdot 10^7$	$5.2 \cdot 10^5$	0.68

kernel	L1 Bound	L2 Bound	L3 Bound	DRAM Bound
ClearParticleClusters	$8.12 \cdot 10^{-2}$	$2.74 \cdot 10^{-3}$	0	$8.07 \cdot 10^{-3}$
ClearParticleGoals	$6.24 \cdot 10^{-2}$	$4.83 \cdot 10^{-3}$	$2.37 \cdot 10^{-2}$	0.5
CountClusters	0.48	$7.07 \cdot 10^{-3}$	$2.99 \cdot 10^{-2}$	0.29
CountClustersPerParticle	0.62	0	$2.03 \cdot 10^{-2}$	$4.84 \cdot 10^{-2}$
CullSmallClusters	0.21	0.12	$1.19 \cdot 10^{-2}$	$3.88 \cdot 10^{-2}$
FracturePotentialSplits	0.15	$5.88 \cdot 10^{-2}$	0.1	0.3
MovingPlaneInteraction	0.29	$3.69 \cdot 10^{-2}$	$7 \cdot 10^{-2}$	0.25
RemoveLonelyParticles	0.24	$2.93 \cdot 10^{-2}$	0.13	0.31
SplitOutliers	$7.11 \cdot 10^{-2}$	$5.36 \cdot 10^{-2}$	0.14	0.3
SplitParticles	$4.11 \cdot 10^{-2}$	$1.04 \cdot 10^{-3}$	$6.16 \cdot 10^{-2}$	0.31
StrainLimitingGoalUpdate	$4.96 \cdot 10^{-2}$	0	$4.21 \cdot 10^{-2}$	0.17
StrainLimitingIteration	$7.86 \cdot 10^{-3}$	$6.95 \cdot 10^{-3}$	0.11	0.13
UpdateClusterComs	0.11	0	0.16	0.12
UpdateClusterPlasticities	$6.72 \cdot 10^{-3}$	$2.33 \cdot 10^{-2}$	$8.22 \cdot 10^{-2}$	0.2
UpdateClusterProperties	$5.47 \cdot 10^{-3}$	$9.82 \cdot 10^{-3}$	$7.84 \cdot 10^{-2}$	0.21
UpdateClusterTransforms	$4.08 \cdot 10^{-2}$	$1.65 \cdot 10^{-2}$	0.19	0.26
UpdateFpAndCom	$6.22 \cdot 10^{-3}$	$2.3 \cdot 10^{-2}$	0.17	0.22
UpdateParticleGoals	$2.58 \cdot 10^{-3}$	$3.08 \cdot 10^{-3}$	$5.03 \cdot 10^{-2}$	$8.58 \cdot 10^{-2}$
UpdateParticleVelocities	$2.03 \cdot 10^{-3}$	$2.03 \cdot 10^{-3}$	0.13	0.15
ZeroParticleGoalPositions	$9.28 \cdot 10^{-2}$	$2.77 \cdot 10^{-3}$	$2.86 \cdot 10^{-2}$	$6.73 \cdot 10^{-2}$

Xeon Profiling Data

Memory

kernel	Memory Bound	Loads	Stores	LLC Miss Count
ClearClusterMembers	$8.48 \cdot 10^{-2}$	0.13	$2.76 \cdot 10^{7}$	$1.73 \cdot 10^{7}$
ClearParticleClusters	0.67	0.56	$4.68 \cdot 10^{7}$	$6.96 \cdot 10^{7}$
ClearParticleGoals	1.47	0.72	$1.49 \cdot 10^{8}$	$1.46 \cdot 10^{8}$
CountClusters	1.79	0.69	$3.96 \cdot 10^{8}$	$3.18 \cdot 10^{8}$
CountClustersPerParticle	0.81	0.39	$3.42 \cdot 10^{8}$	$2.99 \cdot 10^{8}$
CullSmallClusters	$3.43 \cdot 10^{-2}$	0.32	$2.52 \cdot 10^{7}$	$1.9 \cdot 10^{7}$
FracturePotentialSplits	0.53	0.57	$2.26 \cdot 10^{8}$	$1.38 \cdot 10^{8}$
MovingPlaneInteraction	0.85	0.46	$2.86 \cdot 10^{8}$	$2.1 \cdot 10^{8}$
RemoveLonelyParticles	0.26	0.47	$2.12 \cdot 10^{8}$	$8.33 \cdot 10^{7}$
SplitOutliers	0.42	0.24	$3.49 \cdot 10^{8}$	$1.04 \cdot 10^{8}$
SplitParticles	7.29	0.2	$1.49 \cdot 10^{9}$	$3.53 \cdot 10^{8}$
StrainLimitingGoalUpdate	9.53	0.16	$3.92 \cdot 10^{9}$	$1.07 \cdot 10^{9}$
StrainLimitingIteration	2.14	0.26	$2.72 \cdot 10^{9}$	$9.7 \cdot 10^{8}$
UpdateClusterComs	0.99	0.23	$5.36 \cdot 10^{8}$	$1.42 \cdot 10^{8}$
UpdateClusterPlasticities	3.27	0.23	$1.03 \cdot 10^{9}$	$5.35 \cdot 10^{8}$
UpdateClusterProperties	3.04	0.28	$1.56 \cdot 10^{9}$	$9.34 \cdot 10^{8}$
UpdateClusterTransforms	1.05	0.34	$1.35 \cdot 10^{9}$	$4.66 \cdot 10^{8}$
UpdateFpAndCom	1.85	0.24	$5.6 \cdot 10^{8}$	$1.34 \cdot 10^{8}$
UpdateParticleGoals	2.5	0.23	$7.27 \cdot 10^{8}$	$2.37 \cdot 10^{8}$
UpdateParticleVelocities	0.23	0.35	$1.01 \cdot 10^{8}$	$4.97 \cdot 10^{7}$
ZeroParticleGoalPositions	0.31	0.35	$1 \cdot 10^{8}$	$9.16 \cdot 10^{7}$

R9 390 Profiling Data

Memory

kernel	CacheHitPercent	MemUnitBusyPercent	MemUnitStalledPercen
ClearClusterMembers	2.99	0.32	$1.21 \cdot 10^{-2}$
ClearParticleClusters	1.83	2.14	1.78
ClearParticleGoals	29.69	11.16	7.74
CountClusters	18.38	8.3	1.28
CountClustersPerParticle	48.41	37.07	16.97
CullSmallClusters	44.22	10.07	1.66
FracturePotentialSplits	4.53	0.46	$6.44 \cdot 10^{-3}$
MovingPlaneInteraction	31.57	10.24	2.27
RemoveLonelyParticles	1.83	3.56	$8.48 \cdot 10^{-4}$
SplitOutliers	23.4	28.93	11.48
SplitParticles	2.74	3.37	$2.52 \cdot 10^{-3}$
StrainLimitingGoalUpdate	21.1	26.12	5.32
StrainLimitingIteration	17.37	8.14	0.29
UpdateClusterComs	20.23	8.22	0.44
UpdateClusterPlasticities	25.04	10.43	0.68
UpdateClusterProperties	26.84	10.67	0.66
UpdateClusterProperties	26.84	10.67	0.66
UpdateClusterTransforms	17.25	8.11	0.52
UpdateFpAndCom	19.17	8.23	0.67
UpdateParticleGoals	33.67	34.25	11.03
UpdateParticleVelocities	9.3	7.19	1.94
ZeroParticleGoalPositions	1.85	2.1	1.74

ALU Utilisation

kernel	VALUUtilizationPercent	VALUBusyPercent	SALUBusyPercent
ClearClusterMembers	89.61	0	0
ClearParticleClusters	99.94	$8.73 \cdot 10^{-2}$	$8.73 \cdot 10^{-2}$
ClearParticleGoals	99.92	0.11	$9.59 \cdot 10^{-2}$
CountClusters	60.67	$7.21 \cdot 10^{-2}$	$7.62 \cdot 10^{-2}$
CountClustersPerParticle	61.84	0.53	0.52
CullSmallClusters	75	$3.55 \cdot 10^{-2}$	$4.5 \cdot 10^{-2}$
FracturePotentialSplits	48.07	$1.96 \cdot 10^{-3}$	$1.57 \cdot 10^{-3}$
MovingPlaneInteraction	92.29	0.26	0.72
RemoveLonelyParticles	95.36	$9.51 \cdot 10^{-2}$	0.13
SplitOutliers	50.7	0.76	0.77
SplitParticles	96.49	0.17	0.48
StrainLimitingGoalUpdate	61.95	1.49	1.39
StrainLimitingIteration	61.44	0.13	$5.8 \cdot 10^{-2}$
UpdateClusterComs	62.4	$8.73 \cdot 10^{-2}$	$5.69 \cdot 10^{-2}$
UpdateClusterPlasticities	61.95	0.13	$4.8 \cdot 10^{-2}$
UpdateClusterProperties	61.32	0.14	$5.74 \cdot 10^{-2}$
UpdateClusterProperties	61.32	0.14	$5.74 \cdot 10^{-2}$
UpdateClusterTransforms	61.94	0.13	$4.78 \cdot 10^{-2}$
UpdateFpAndCom	62.39	$8.74 \cdot 10^{-2}$	$5.71 \cdot 10^{-2}$
UpdateParticleGoals	62.5	0.87	0.64
UpdateParticleVelocities	99.92	0.13	$6.79 \cdot 10^{-2}$
ZeroParticleGoalPositions	99.94	$9.62 \cdot 10^{-2}$	$6.83 \cdot 10^{-2}$

Bibliography

[1] OpenCL conformant products. `https://www.khronos.org/conformance/adopters/conformant-products#opencl`. Accessed: 09-07-2016.

[2] OpenCL version 1.2 specification. `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`. Accessed: 09-07-2016.

[3] A. W. Bargteil and B. Jones. Strain limiting for clustered shape matching. In *Proceedings of the Seventh International Conference on Motion in Games*, MIG '14, pages 177–179, New York, NY, USA, 2014. ACM.

[4] J. C. Bezdek, R. Ehrlich, and W. Full. Fcm: The fuzzy c-means clustering algorithm. *Computers & Geosciences*, 10(2-3):191–203, 1984.

[5] Z. Chen, M. Yao, R. Feng, and H. Wang. Physics-inspired adaptive fracture refinement. *ACM Trans. Graph.*, 33(4):113:1–113:7, July 2014.

[6] G. Debunne, M. Desbrun, M.-P. Cani, and A. H. Barr. Dynamic real-time deformations using space & time adaptive sampling. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 31–36. ACM, 2001.

[7] J. C. Dunn. A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. *Journal of Cybernetics*, 3(3):32–57, 1973.

[8] W.-c. Feng and S. Xiao. To gpu synchronize or not gpu synchronize? In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 3801–3804. IEEE, 2010.

[9] A. Greß, M. Guthe, and R. Klein. Gpu-based collision detection for deformable parameterized surfaces. In *Computer Graphics Forum*, volume 25, pages 497–506. Wiley Online Library, 2006.

[10] B. Jones, A. Martin, J. A. Levine, T. Shinar, and A. W. Bargteil. Clustering and collision detection for clustered shape matching. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, MIG '15, pages 199–204, New York, NY, USA, 2015. ACM.

[11] B. Jones, A. Martin, J. A. Levine, T. Shinar, and A. W. Bargteil. Ductile fracture for clustered shape matching. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '16, pages 65–70, New York, NY, USA, 2016. ACM.

[12] J. A. Levine, A. W. Bargteil, C. Corsi, J. Tessendorf, and R. Geist. A peridynamic perspective on spring-mass fracture. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Jul 2014.

[13] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[14] M. Müller, N. Chentanez, and T.-Y. Kim. Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph.*, 32(4):115:1–115:10, July 2013.

[15] 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*, SCA '02, pages 49–54, New York, NY, USA, 2002. ACM.

[16] M. Müller and M. Gross. Interactive virtual materials. In *Proceedings of Graphics Interface 2004*, GI '04, pages 239–246, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[17] M. Müller, B. Heidelberger, M. Teschner, and M. Gross. Meshless deformations based on shape matching. *ACM Trans. Graph.*, 24(3):471–478, July 2005.

[18] A. Norton, G. Turk, B. Bacon, J. Gerth, and P. Sweeney. Animation of fracture by physical modeling. *The Visual Computer*, 7(4):210–219, 1991.

[19] 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, pages 165–175, New York, NY, USA, 2009. ACM.

[20] S. Raghavachary. Fracture generation on polygonal meshes using voronoi polygons. In *ACM SIGGRAPH 2002 conference abstracts and applications*, pages 187–187. ACM, 2002.

[21] A. R. Rivers and D. L. James. Fastlsm: Fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph.*, 26(3), July 2007.

[22] S. C. Schvartzman and M. A. Otaduy. Fracture animation based on high-dimensional voronoi diagrams. In *Proc. of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2014.

[23] R. Spliet, L. Howes, B. R. Gaster, and A. L. Varbanescu. Kma: A dynamic memory manager for opencl. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 9:9–9:18, New York, NY, USA, 2014. ACM.

[24] D. Terzopoulos and K. Fleischer. Modeling inelastic deformation: viscolelasticity, plasticity, fracture. In *ACM Siggraph Computer Graphics*, volume 22, pages 269–278. ACM, 1988.

[25] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. In *ACM Siggraph Computer Graphics*, volume 21, pages 205–214. ACM, 1987.