Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Acceleration of Physics Simulation Engine through OpenCL

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Jorge Miguel Raposeira Lagarto

Orientador: Prof. Doutor Fernando Birra

Maio de 2011

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Acceleration of Physics Simulation Engine through OpenCL

Jorge Miguel Raposeira Lagarto

Orientador: Prof. Doutor Fernando Birra

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

Maio de 2011

Acceleration of Physics Simulation Engine through OpenCL

Resumo

Hoje em dia, a simulação física é um tópico relevante em vários domínios, desde áreas científicas como a medicina, até ao entretenimento como efeitos em filmes, animação de computador e jogos.

Para facilitar a produção de simulações mais rápidas, os programadores usam motores de física, pois eles oferecem uma grande variedade de funcionalidades como simulação de corpos rígidos e deformáveis, dinâmica de fluídos ou detecção de colisões. As indústrias do cinema e dos jogos de computador usam cada vez mais os motores de física para introduzir realismo nos seus produtos. Nestas áreas, a velocidade é mais importante do que a precisão e têm sido desenvolvidos esforços para atingir simulações com alta performance. Além de algoritmos de simulação física mais rápidos, o avanço na performance das Unidades de Processamento Gráfico (GPUs) nos últimos anos tem levado os programadores a transferir cálculos pesados para este tipo de dispositivos, em vez de os realizar na Unidade Central de Processamento (CPU). Alguns motores de física já fornecem implementações em GPU de algumas funcionalidades, nomeadamente na detecção de colisões entre corpos rígidos.

Neste trabalho pretendemos acelerar uma funcionalidade presente na grande parte dos motores de física: a simulação de tecidos. Como a detecção de colisões é um dos maiores factores que limitam a eficiência neste tipo de simulação, vamo-nos concentrar especificamente em melhorar esta fase. Para atingir uma aceleração considerável, planeamos explorar o incrível paralelismo do GPU através do desenho de algoritmos eficientes na arquitectura *Open Computing Language* (OpenCL). Finalmente, irá ser elaborado um estudo para comparar a performance de uma implementação sequencial em CPU com a solução paralela apresentada em GPU.

Palavras-chave: Motor de física; Simulação de tecidos; Detecção de colisões; GPU - Unidade de Processamento Gráfico; OpenCL - *Open Computing Language*; BV - Volume envolvente; BVH - Hierarquia de volumes envolventes

Abstract

Nowadays, physics simulation is a relevant topic in several domains, from scientific areas like medicine to entertainment purposes such as movie's effects, computer animation and games.

To make easier the production of faster simulations, developers are using physics engines because they provide a variety of features like rigid and deformable body simulation, fluids dynamics and collision detection. Computer game and film industries use increasingly more physics engines in order to introduce realism in their products. In these areas, speed is more important than accuracy and efforts have been made to achieve high performance simulations. Besides faster physical simulation algorithms, GPUs' performance improvement in the past few years have lead developers to transfers heavy calculation work to these devices instead of doing it in the Central Processing Unit (CPU). Some engines already provide GPU implementations of several key features, particularly on rigid body collision detection.

In this work we want to accelerate a feature present in most of the current physics engines: cloth simulation. Since collision detection is one of the major bottlenecks in this kind of simulation, we will focus specifically in improving this phase. To achieve a considerably speed-up we plan to exploit the massive parallelism of the Graphics Processing Unit (GPU) by designing an efficient algorithm using the Open Computing Language (OpenCL) framework. Finally, a study will be made to compare the performance of a sequential CPU approach against the parallel GPU proposed solution.

Keywords: Physics engine; Cloth simulation; Collision detection; GPU - Graphics Processing Unit; OpenCL - Open Computing Language; BV - Bounding Volume; BVH - Bounding Volume Hierarchy

Contents

List of Figures

List of Tables

Acronyms

AABB	Axis-Aligned Bounding Box.
API	Application Programming Interface.
BV	Bounding Volume.
BVH	Bounding Volume Hierarchy.
CCD	Continuous Collision Detection.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
DOP	Discrete Oriented Polytope.
GPU	Graphics Processing Unit.
LBVH	Linear Bounding Volume Hierarchy.
OBB	Oriented Bounding Boxes.
OpenCL	Open Computing Language.
PPU	Physics Processing Unit.
SAH	Surface Area Heuristic.
SM	Streaming Multiprocessor.
SP	Streaming Processor.

1. Introduction

Physics is a huge area of investigation, which combined with computer science, is used to simulate aspects of the physical world. From this union results the physics engine, a software capable of performing an approximate simulation of the real world, based on physics laws. Rigid and soft body dynamics, collision detection and fluid simulation are some of the features offered by this kind of software.

Physics engines are used in a variety of domains, from scientific research to computer games or movie's special effects. While some areas need high accuracy, which leads to expensive computations, in others speed of simulation is more important than precision. In computer game and film industries, physics engines usually resort to simplified calculations, lowering their accuracy and accelerating the simulation, providing real time gameplay and continuous animation.

To meet this demand for speed, physics engines began to explore the power of actual GPUs (Graphics Processing Unit). In the past few years, GPUs have improved their performance dramatically, when compared to CPUs. This gap does not have to do with processor's clock speed-up, but with the difference in the number of cores between CPUs and GPUs. Such a high number of cores results on a considerable increase in the amount of work a GPU performs at once, by doing tasks in parallel. Each core can process large data sets by running thousands of threads at the same time.

To help developers take advantage of these capabilities, NVIDIA launched Compute Unified Device Architecture (CUDA) in 2007, providing an easy way to access cores and execute code in parallel. Before CUDA, it was very difficult to use GPUs because the processor cores were only accessible through Application Programming Interfaces (APIs) oriented towards common graphics tasks. However, CUDA is limited to NVIDIA's GPUs, so in 2008 OpenCL was released, offering a standard platform.

Nowadays, some physics engines are already using GPU acceleration through CUDA or OpenCL in features like rigid body collision detection and particle interaction. This work will focus on cloth simulation, which is a very important topic in areas like computer animation, textile industry and games. Specifically, we will accelerate cloth collision detection using the GPU to perform expensive computations.

This option was chosen after analyzing the features of three physics engines (PhysX, Havok Physics and Bullet). The conclusion was that currently, there is little work done on this domain using GPUs despite its enormous potential for improvement by doing tasks in parallel.

A cloth mesh is composed by a large number of triangles. During the simulation, each triangle is in motion, which can cause intersections with other objects' triangles (inter-object collision) or even with triangles in the same mesh (self-collisions). In order to correctly detect all the collisions, we need to perform a huge amount of tests which can lead to several pairs of triangles to examine. Having the ability to use the GPU, we can run all these tests in parallel and improve substantially the collision detection phase performance.

1.1 Motivation

The main motivation for this work is the possibility to improve physics simulation performance by exploiting the capabilities of current GPUs, coupled with the utilization of OpenCL has a generic way to produce parallel code.

Cloth collision detection was the feature chosen to be accelerated, due to the importance of cloth simulation in physics engines, joined with the fact that collision detection in this area remains little explored concerning to GPU utilization. Moreover, collision detection is an expensive process because of the large number of elementary tests needed between triangles to find intersections. These tests are the strong candidates to be parallelized by several GPU cores.

Spatial enumeration techniques are not attractive to be applied in the context of collision detection involving deformable objects, as they would require to constantly update the spatial data structure. Instead, bounding volume hierarchies provide an easier alternative. Cloth triangles are subdivided in areas and organized in a tree hierarchy. Nearby triangles (enclosed in the same subdivided area) are stored in the same node to facilitate collision detection. To discover collisions between two mesh based objects we need to traverse and compare their hierarchies and test pairs of possible colliding triangles. The goal is to do this traversal, as well as the elementary tests involving triangles in parallel by running OpenCL kernels on the GPU.

We use as model a cloth simulator developed in [6]. The main intention is to transfer the expensive phase of collision detection (or the more relevant parts) to the GPU and improve the simulator performance.

1.2 Expected contributions

With this work we expect to accelerate a cloth simulator [6] by improving its collision detection system. To accomplish this task, we plan to transfer most of the calculations from the CPU to the GPU using the OpenCL framework.

In summary, the main expected contributions are:

- Design and implementation of parallel algorithms through OpenCL architecture for collision detection on cloth simulation, capable of running in the GPU with minimal intervention from the CPU.

- Development of a comparative study in terms of speed-up between the results obtained from the proposed solution and the studied approaches, as well as a comparison between both CPU sequential and GPU parallel implementations.

2. State of Art

This chapter is divided in three phases. First, we will focus on actual physics engines, specifically on the three major contributors in this area: PhysX, Havok and Bullet. The primordial goal is to analyze the main features of each one with special attention to developments made in GPU parallel computation and in the end do an overall comparison.

In the second phase we explain in more detail the physics simulation feature that we want to accelerate through the GPU: cloth collision detection. Particularly, we will describe techniques used to efficiently detect collisions in cloth meshes and the advances made using the GPU.

Finally, we will give an overview about the OpenCL framework used to produce parallel algorithms capable of running on the GPU.

2.1 Physics Engines

Computer game and film industries use increasingly more physics engines in order to introduce realism in their products. To meet these demands, physics engine developers work to refine simulation techniques to incorporate in their computer's software. Rigid and soft body dynamics, fluid dynamics or collision detection are examples of physical phenomena capable of being simulated in physics engines. However, in most cases, speed is more important than accuracy and efforts have been made to achieve high performance simulation. Besides faster physical simulation algorithms, GPU's performance improvement in the past few years have lead developers to transfer heavy computation work to these devices instead of doing it on CPU.

This section focuses on actual physics engines, specifically on the three major contributors in this area: PhysX, Havok and Bullet. We will give a brief overview of these engines and focus special attention to the developments made in GPU parallel computation.

2.1.1 PhysX

PhysX [4] is NVIDIA's proprietary physics engine that provides support for physical simulations used in actual PC and console games. It has a free and closed source version available in NVIDIA's home page, but developers can pay to obtain full source code. *Medal of Honor: Airborne*, *Unreal Tournament 3* and *Batman: Arkham Asylum* are examples of realistic games running with this engine.

PhysX provides a set of features covering the main aspects of physic simulation, from rigid to soft bodies and cloth, collision detection or fluids. In addition to its wide range of features, it introduces innovations in cloth tearing and soft body collision detection with fluids. Almost all of its functionalities are capable to run totally on NVIDIA's Physics Processing Unit (PPU) hardware and it also gives the possibility to parallelize some computations on GPU through CUDA framework. However, it has the disadvantage of being a closed source SDK, which does not confer great customization. As a proprietary system, it also restricts the hardware acceleration to NVIDIA's PPUs or GPUs, not offering a scalable service concerning hardware parallelization yet.

2.1.2 Havok

Havok Physics [3] is one of the most popular physics engines in game and film industries, developed by a company called Havok. It gathers a set of features capable of running across multiple platforms like PC, Xbox 360, PlayStation 3 or Nintendo's Wii. It also provides solutions for digital media creators in the movie industry, helping film production, including *The Matrix*, *X-Men: The Last Stand* or *Charlie and the Chocolate Factory*. Havok Physics is integrated in a suite of tools developed by Havok, each one with different features and directed to a specific goal.

Havok Physics itself does not provide a rich kit of solutions. It is limited to rigid bodies and collision detection. For a full utilization of its capabilities we have to integrate it with the remaining Havok products. It has an efficient collision detection mechanism and a continuous simulation system for fast object contact point's recognition. Havok is fully multi-threaded for CPU or Playstation's Cell SPU and offers great flexibility because of its open source SDK. Unfortunately, so far it was not possible to take benefit from GPU optimizations, despite an OpenCL demonstration on Havok Cloth.

2.1.3 Bullet

Bullet [2] is a freeware and open-source physics library. Its SDK is implemented in C++ language and is used in game development and movie special effects. Bullet is used in a variety of game platforms and has optimized multi-threaded code for Cell SPU, CUDA and the newest OpenCL.

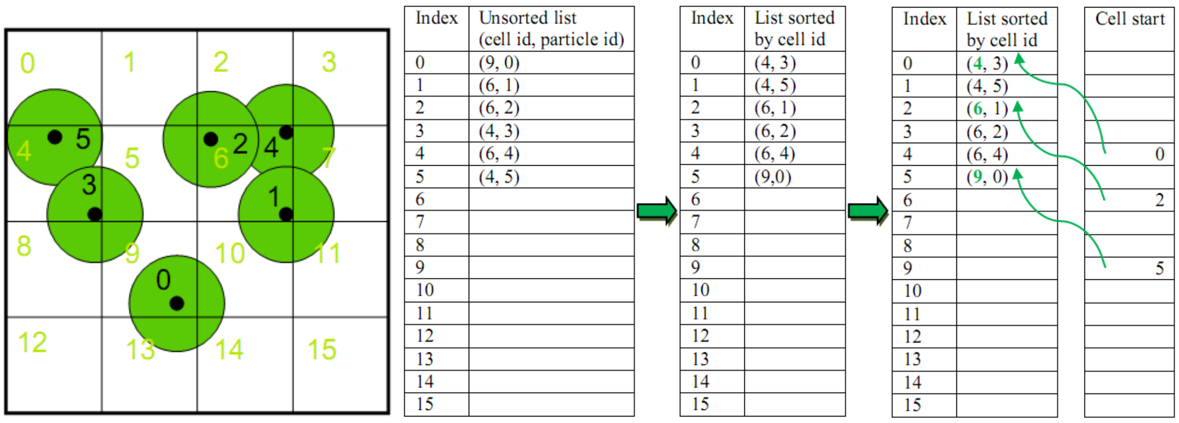
The SDK gathers a collection of good features, such as rigid and soft body simulation, discrete and Continuous Collision Detection (CCD). Its C++ source code gives the freedom to customize engine's features or port them to different platforms. However, Bullet's most promising feature is GPU acceleration. Besides a CUDA broad phase algorithm, developers are working on OpenCL collision detection and a constraint solver, allowing for a bigger hardware scalability.

2.1.4 GPU Acelleration in Physics Engines

CUDA is used in PhysX and Bullet engines to accelerate rigid body collision detection and particle based applications where every particle interacts with each other (for example, through gravitational attraction). For systems where particles don't interact with each other, parallelization is relatively easy because each particle can be treated independently, but to run a simulation where interparticle forces are considered, a refined approach is needed. To cover all the particle interactions, in a brute-force implementation, we need to perform $n(n-1)/2$ collision tests for *n* particles. However, we can take advantage of the fact that the interaction force drops off with distance and comparisons will only be needed between each particle and its neighbors, using a spatial subdivision method. The same can be applied to rigid body collision detection, since only nearby bodies have a chance to collide.

The implementation described in [12], is used in PhysX and Bullet and consists in partitioning the simulation space into a uniform grid (Figure 2.1), where the cell size is equal to the particle size (for different particle's sizes, is the same as the largest size), so each object can occupy a maximum of 2^d cells, where d is the number of dimensions (2^3 cells in a 3D world). With this cell size, only four tests are needed for a 2D object and eight for a 3D object collision detection. The cell id on the grid is calculated for each particle based on its center points (for larger objects, it could be based on their bounding boxes positions). From this cell id, a hash value is computed to parameterize object cell position to CUDA's grid. The cell hash value is stored along with the object id in an array which will be further transferred to the GPU. Once in the GPU, this array is sorted by cell hash value, using a parallel radix sort algorithm, described in detail by Scott Le Grand [11].

The next phase consists in signaling the cells containing particles in order to help the collision detection in the next step. This is achieved by writing in a new array (cell start array) the indexes in the sorted array where new cells appear. As shown in Figure 2.1, cells 4, 6 and 9 start on indexes 0, 2 and 5 respectively in the sorted array. To fill the array, threads (one per particle) gets the particle's cell index in the ordered array and compares it with the cell index of the previous particle. If the indexes differ, it means that a new collision cell was found and the index is written.



Index	Unsorted list (cell id, particle id)
0	(9, 0)
1	(6, 1)
2	(6, 2)
3	(4, 3)
4	(6, 4)
5	(4, 5)
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

Index	List sorted by cell id
0	(4, 3)
1	(4, 5)
2	(6, 1)
3	(6, 2)
4	(6, 4)
5	(9,0)
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

Index	List sorted by cell id	Cell start
0	(4, 3)	
1	(4, 5)	
2	(6, 1)	
3	(6, 2)	
4	(6, 4)	0
5	(9, 0)	
6		2
7		
8		
9		5
10		
11		
12		
13		
14		
15		

Figure 2.1 Collision detection on the GPU: left image shows an example of bodies disposition on a 2D grid; right image shows how the array containing the particles and their cells on CUDA is before and after radix sort algorithm and how cell start array is filled

In the final stage, each thread obtains one particle id from the ordered array, finds its Axis-Aligned Bounding Box (AABB) in an AABB's array and calculates in which cell (in the world's grid) the particle is in. Then it examines the neighboring 27 ($3{\times}3{\times}3$) cells and for each one computes its hash value (for CUDA's grid) and checks in the new filled array (cell start) if there are particles in that cell id. If the cell is not empty (contains an integer index representing its index in the ordered array), iterates over all bodies in the ordered array beginning on the index read from cell start array and tests for collisions.

Since this implementation is in CUDA architecture, developers are still restricted to NVIDIA's GPUs. To achieve greater scalability, NVIDIA is considering the use of OpenCL in future PhysX releases.

However, OpenCL is already incorporated in Bullet's SDK. Since version 2.77 (September 2010), one of Bullet's features is the soft body and cloth library translated to OpenCL. So far, the current implementation only parallelizes the integration phase, forces computation and constraint solver and does not provide support fro collision detection or other advanced features.

We can also check the Bullet's OpenCL broad phase mechanism that uses a parallel Bitonic Sort algorithm for bodies' hash sort, different from CUDA's Radix Sort implementation. Bitonic Sort [16] is slower than Radix Sort [13, 26] for large data sets but performs better when has a small number of elements to sort. Moreover, for short arrays that can fit on GPU shared memory, execution is performed in a single kernel and take advantage of local memory fast access.

Presently AMD supports Bullet's OpenCL developments, so it is expectable to see progress in this area, and in the meantime, we can follow the latest implementation updates on Bullet's SVN [1].

2.2 Cloth Collision Detection

A cloth mesh is composed by a set of vertices, connected by edges, forming faces (usually triangles). Due to cloth complex structure and deformation capacity, collision detection is more difficult than for simple rigid bodies. To successfully identify collisions between cloth meshes and generic mesh objects (deformable or rigid), as well as cloth self intersections, two tests are needed (Figure 2.2), as demonstrated by Provot [25]:

- One vertex from one mesh intersects a face from another mesh (vertex/face collision).
- One edge from one mesh intersects an edge from another mesh (edge/edge collision).

Once more, testing all cloth triangles (vertices, edges and faces) against all obstacles' triangles is not the best approach. It is a very time consuming solution and leads to unnecessary tests between distant triangles that have no chances to collide. To deal with this problem and accelerate the collision detection phase performance we have to think in a more efficient strategy.

We will focus on two methods to improve cloth collision detection performance: Spatial Enumeration and Bounding Volume Hierarchies combined with space partition. In the following sections these approaches will be analyzed based on their advantages and disadvantages.

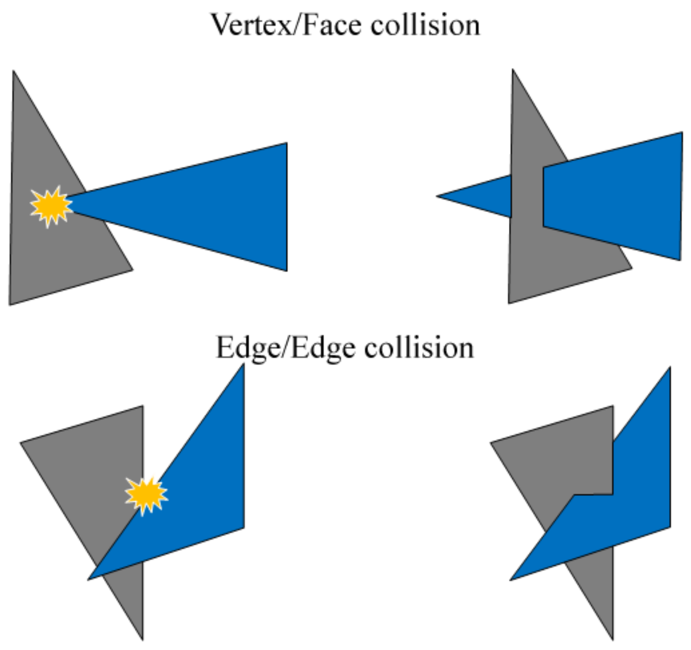


Figure 2.2 Collision types in cloth meshes: Vertex/Face collision and Edge/Edge collision

2.2.1 Spatial Enumeration

This technique consists in dividing the 3 dimensional simulation space, into smaller volume element boxes and assign each primitive (triangles for mesh based objects) to one of the boxes (based on triangle's center point), allowing to check collisions only between primitives contained in the same or neighboring boxes. The grid is fixed during the simulation and we have to compute objects' new cells within the grid at each time step. The volume organization generally varies between regular grids and hash tables (spatial hashing).

2.2.1.1 Regular Grids

We saw this technique applied by PhysX and Bullet engines for rigid body collision detection. Each object was assigned to one grid's box, limiting each object's collision test to the neighboring cells. However, to achieve this optimization we needed to choose the right cell size. If the box was too small, a triangle could intersect too many cells and increase the number of cells to test. On the other hand, an excessively large cell would contain too much triangles, generating useless tests involving far away primitives. The best choice was to have box dimensions equal to object's dimensions. This way a triangle can overlap a maximum of 8 cells and collisions test can be resumed to the 27 surrounding boxes. Although this approach works very well for rigid bodies, it has a problem when it comes to soft body collision detection. In soft body simulation, triangles change their aspect frequently due to forces acting on them (stretching, bending, etc.) making it impossible to predict their exact size. Very large cells can eventually solve this, but also causes excessive triangles per box, decreasing the method's performance.

The usual solution is to store each triangle in every cell it overlaps, creating a data structure larger than the initial set of triangles. A bounding volume is computed for each triangle (usually an AABB due to its simplicity) to easily identify the overlap cells. Then, in a second phase, we compute again each triangle's overlapped cells through its AABB and check for intersection with other triangles in the same boxes. If two primitives overlap, a collision is detected and if they belong to the same object a self-collision is found.

Regular grids are simple and efficient, but they are not adaptive, which could result in many empty cells or many over-populated cells. This issue can be slightly improved with spatial hashing, since the hash table size is normally smaller then the whole grid.

2.2.1.2 Spatial Hashing

Spatial hashing [28] works by storing mesh vertices in a hash table (spatial hashing). Each vertex has a key, corresponding to its grid box identifier. Based on this key a hash value is calculated to obtain the hash table index where the vertex will be inserted. If two vertices with different key values obtain the same hash value, the insertion function will iterate over the hash table until it finds an empty position to keep one of the elements, which means that each array location will have vertices with identical keys (contained in the same cube). After filling the array with all vertices, starts the second phase of collision detection. All triangles are iterated and their AABB's computed to discover the cells affected by the bounding volume. For each cell overlapped by the AABB, its hash value is determined and intersection tests are made between triangle and vertices in the same index (vertex/triangle test), as well as all the edges connected to the vertex (edge/edge test).

2.2.2 Bounding Volume Hierarchies

The basic idea is to partition the space occupied by the objects' bounding volumes hierarchically, storing it in a tree structure called Bounding Volume Hierarchy (BVH), used both for inter-object and self collision detection.

Each BVH refers to one mesh object, which means that to check if two deformable bodies are colliding we need to compare their BVHs. For a large number of bodies an exponential number of BVH evaluations will be needed, which is very time consuming and inefficient. One way to reduce that amount of tests is to enclose all objects with BVs and treat them like rigid bodies. Then it is possible to apply techniques like regular grids to check only the closest meshes.

2.2.2.1 Bounding Volume choice

Bounding Volume (BV) are closed volumes built around primitives that are used to accelerate collision detection. Instead of testing every possible combination of two colliding triangles, one from each object (or a smaller partition of an object), BVs allow to verify only if their volumes intersect with a few tests, depending on BV's complexity. The most well known BVs for collision detection are Bounding Sphere, AABB, Oriented Bounding Boxes (OBB) and *k*-Discrete Oriented Polytope (DOP), each one with different characteristics. Figure 2.3 illustrates the equivalent 2D representations of these bounding volumes.

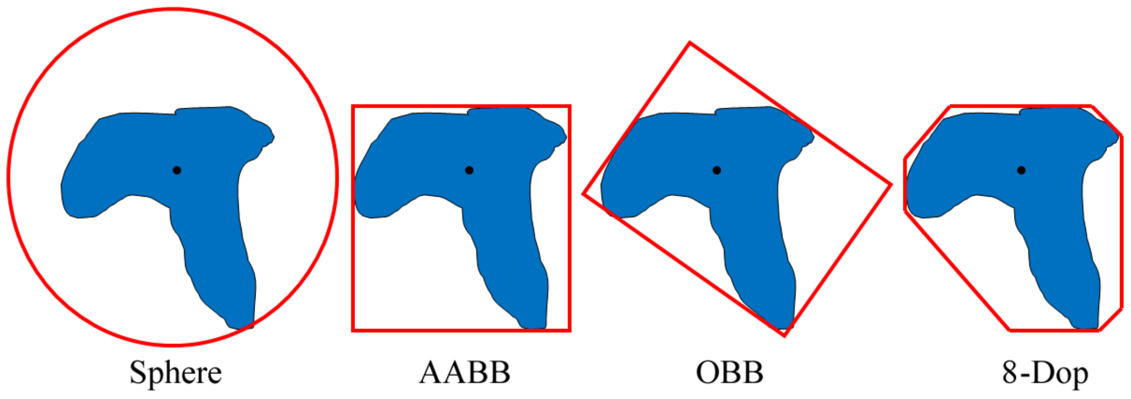


Figure 2.3 Bounding Volumes: Sphere, AABB, OBB and 8-Dop

Bounding Sphere [24] is the simplest BV and the easier to test for intersection. It is defined by a sphere centered on the object's center with radius equal to the distance from the body's center and its farthest vertex.

AABBs [29] and OBBs [10] are rectangular boxes that completely contain an object. An AABB is a box aligned with the coordinate axes and it only needs two tests per axis (six on total) to detect a collision with another AABB. OBB has an arbitrary alignment, frequently with object's alignment for tighter fit. Collision detection is slower than for AABB but its update is relatively easier if the object just moves without altering its size (good for rigid bodies) because its OBB simply needs a rigid body transformation to follow body's movement.

Another common type of bounding volume is the *k*-DOP. DOP stands for Discrete Oriented Polytope, which is the equivalent to a convex polygon in three dimensional space and the value of *k* determines how many faces it has. An AABB (in 3D) is equivalent to a 6-DOP, having the planes' normals oriented by the three dimensional axes. Values of *k* typically vary between 6, 14 18 and 26, which is reflected on BV's tightness. To detect an overlap between two *k*-DOPs only *k* tests are necessary, at most, and the update cost per leaf node is proportional to the number of vertices inside the DOP, while it is in the order of *k* for internal nodes.

Bounding volume choice is an important issue in order to optimize the queries. We must decide based on a good balance between performance (time required to perform an overlap test) and accuracy (quantity of false collisions eliminated in each test). A simpler and less tight BV such as AABB, OBB or Sphere bounding volume, guarantees faster and easier overlap tests, as well as lower memory requirements when compared to tighter ones like *k*-DOPs. On the other hand, more complex BVs can reduce the number of potential colliding triangle candidate pairs and consequently the number of elementary tests (Vertex/Face and Edge/Edge) between triangles.

According to [17], k-DOPs offer the best solution due to their tight volume that limits the number of unnecessary elementary tests, paying off construction and update costs. In comparison, OBBs also provide a decent approximation to the set of primitives but have a prohibitive update cost. Nevertheless, in [17] it's also referred that k-DOPs should be composed by pairs of parallel planes, bounding the primitives along $k/2$ directions. This allows to compute k-DOPs using simple dot products between object's vertices and the k planes' normal vectors, minimizing the associated costs.

2.2.2.2 BVH Construction

Every mesh's BVH is built in a pre-processing step, usually using a top-down approach. The BV envolving the entire mesh will match the tree's root and its descendants result from successive splits in the ancestors' BV. Usually, the arity of the tree depends on the number of axes split. For instance, one axis split results in a binary tree, two axes in a 4-ary tree (quadtree) and three axes in a 8-ary tree (octree). If we don't want to divide along all the axes, the best option is often to cut where the BV is longer, in general on the BV's center because it guarantees the smaller child volumes. Triangles intersected by the splitting plane are placed on the side where they have their centroid, or in alternative positioned on the BV with fewer primitives.

This recursive technique stops when it was reached a defined threshold on the quantity of triangles per volume. Besides its BV definition, a leaf node will also contain information about the triangles included in its volume.

2.2.2.3 BVH Traversal

In order to find possible collisions between two objects we must recursively traverse their BVHs and check for BV intersections, starting on both roots. Whenever a non-overlapping node pair is reached, the algorithm stops its recursive traversal in that branch. However, if an overlap is found and both nodes are leafs, their enclosed triangles are tested for intersection. If only one node is leaf, the following tests are between the leaf and the internal node's children. Lastly, if the two nodes are internal, we choose one to step down and test its children. The best decision is to examine the node with smaller volume against the children of the node with larger volume (the decision can also be based on each node's height in the tree or in the number of descendants), with the aim of quickly reach a stopping point in the algorithm (leafs intersection or no intersection at all).

Although this approach produces the right results, it's possible to minimize the number of BV overlap tests, reducing this way the collision detection time. A technique introduced by Klosowsky in [17] exploits temporal coherence between successive frames to improve queries performance. The main idea is to keep track of the last (deepest) node pairs (one node from each tree) tested for overlap in the previous collision detection query, storing them in a *front* data structure and start examining from there at the next time step. This can be advantageous considering that if two BVs intersect each other at some instant in time, then it's quite likely that they (or the BVs nearby) also overlap after a small interval of time.

At the beginning, the front will contain only the roots from the two hierarchies (last nodes traversed), but it will be updated during the simulation. It can be *dropped*, when a node pair in the front is replaced by one of its descendants: the last traversed until a non-overlapped node pair is reached or a pair of leaf nodes, meaning that a possible collision was found. The other option is to do a *raise* operation, required when a particular node pair corresponding to an overlap between two BVs at the previous instant, no longer represents an intersection. In these cases, the node pair is removed from the front and it's replaced by one of its ancestors (the first matching an intersection). To find the replacement pair, we choose one of the nodes from the removed pair to climb (usually the deepest in its hierarchy).

2.2.2.4 BVH Update

At each time step, a cloth object can move, deform or collide with another object, affecting its mesh arrangement and, consequently its BV and tree representation. This means that the hierarchy needs to be updated frequently to keep up with changes.

A "brute force" approach is to recompute the entire BVH at each discrete time step, which has obviously an exaggerated cost ($O(n \log n)$ for n triangles) and will slow down significantly the simulation.

A better solution [20] is to refit the bounds of each BV in a bottom-up manner ($O(n + \log n)$ for n triangles). Starting on leaf nodes, all triangles are checked to discover if their movement has placed their vertices out of the BV. In positive cases the BV is updated to fully enclose the triangles and changes are propagated up in the tree towards the root. Each BV is merged with its siblings and the result is used to refit their parent's BV. We can reduce even more the update time by stopping refitting when the merged BV of all children is completely inside of their parent's BV, before reaching the root.

However, successive updates result on a degradation of BVH performance in collision queries because primitives change their positions but the structure remains the same, leading to less tight fit BVs (with more empty space that could guide in false intersections). This aspect is specially noted when the cloth mesh suffers significant modifications during simulation time. To rectify the hierarchy and guarantee more efficient collision detections, we have to perform entire rebuilds from time to time. A simple approach is to rebuild the tree in pre-defined and fixed time intervals or number of steps, but it has a drawback: it is not adaptive to mesh deformations. The process used in [8] compares a node to its children based on their volumes and decides to rebuild the hierarchy when the ratio between them becomes very large.

Alternatively, we can just update a part of the tree and leave the remaining nodes out of date until we need them (lazy update). The method proposed by Larsson [18] opts to update only the upper half of the tree (starting halfway ($height/2$) proved to be the best after the results achieved in several experiments). The first BVs (at the halfway of the tree) are refitted based on the set of triangles contained in their sub-trees and the BVs above them (their ancestors) are computed using the merging method explained earlier. This update system is faster than normal bottom-up update if we can enlarge the BV based on a set of vertices instead of triangles. This is explained by the fact that lower level nodes have more vertices shared by various triangles that are stored in more than one BV. Larsson refers that level $d = height/2$ requires approximately $n/2$ vertex tests for n triangles, in other words, half the time needed to refit the BV when compared to leaf nodes. Then, the outdated nodes are repaired in the collision detection phase as they are needed. If the traversal reaches one of these nodes for an overlap test, the BV is recalculated in a top-down manner by splitting its parent BV. Obviously, this will result in some loss of time gained in the update phase.

Nevertheless, the same author improves this technique by introducing a dynamic update scheme that takes advantage of temporal coherence between successive frames in cloth simulation [19]. Instead of updating the upper half of the tree, the algorithm starts at the deepest nodes visited in the last collision detection query. The refitting is done the same way: the first BVs are recalculated based in their descendant set of primitives, the nodes above are built in a bottom-up style by merging child BVs and the outdated nodes are constructed during the collision detection phase. This scheme is similar to the front based mechanism used for collision detection improvement, providing a more accurate starting point for updating.

2.2.3 GPU Implementations

As it can be seen, collision detection involves many computations to correctly discover and treat primitive intersections, as well as time spent updating BV data structures, preparing them for the next collision query. This results in a major bottleneck in cloth simulation and GPU utilization emerges as an interesting option to accelerate its performance. Actual GPUs provide a large number of cores, capable of running a huge number of threads in parallel. These properties can be exploited by designing the right algorithms, choosing the best tasks to execute in parallel and the optimal way to distribute them.

Cloth collision detection has several phases that can be parallelized on GPU. One of them is BVH construction and is addressed in [7] where the authors describe two different construction algorithms.

The first one, called Linear Bounding Volume Hierarchy (LBVH), uses an efficient method based on Morton Codes to build hierarchies. The codes are calculated from primitives' geometric coordinates and used to map each primitive to its correspondent BV. First, a $2^k \times 2^k \times 2^k$ grid is established from the enclosing AABB of the entire geometry. Once this grid is made, a $3k$-bit Morton Code is assigned to each cell by interleaving the binary coordinate values (Figure 2.4).

It is now possible to select the right grid cell for each primitive based on their coordinates and give them the correspondent code. After this initialization phase, begins the hierarchy construction by ordering the primitives' codes. At the first level we look for the most significant bit of all codes and place those with 0 and 1 in the left and right child respectively. The process of subdivision is then repeated by examining the next bit until all the $3k$-bits are processed. This partitioning method is highly parallel because it uses a fast radix sort [13, 26] to split the primitives.

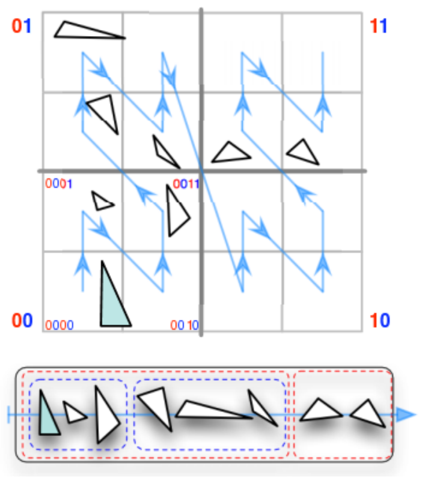


Figure 2.4 Example of 2D Morton Code grid.

The second method uses the Surface Area Heuristic (SAH) to construct BVHs optimized for ray-tracing. The SAH consists in evaluating all possible split positions among all split planes and choose the one with lowest cost according to the ratio between its surface area and the number of primitives within it (Figure 2.5).

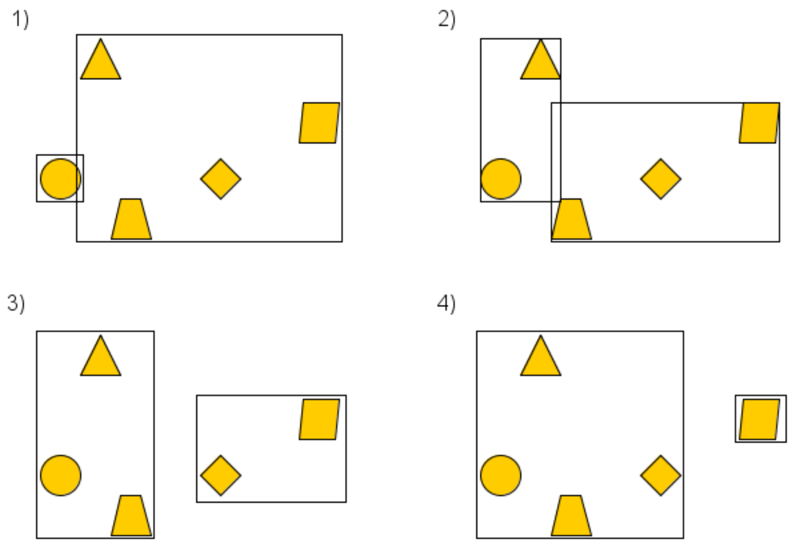


Figure 2.5 Possible object partitions along a single coordinate axis. SAH method tries to find the minimum total volume.

After selecting the split coordinate, the algorithm sorts the primitives into the left or the right node, depending on their centroid, through a parallel prefix sum operation [13]. Once sorted, one thread updates the node's bounding box information. In order to parallelize these steps, the authors propose a working queue approach to distribute the splits across all cores. Each block of threads reads a split object from the input queue, processes it (SAH evaluation, primitive sorting and BV update) and writes the resulting splits in the output queue. By using two work queues, one as input and another as output, in each construction level, we can avoid synchronization between them. In a binary tree the output queue should have two times the size of the input queue because we know that a split item can result at most in two new nodes. However, sometimes a split generates only one (one is leaf) or even zero new nodes (both leafs). These leaf nodes are written in the output queue as null objects because they don't need to be divided in the next level, which forces a compaction before the next split kernel to ensure that only non leaf nodes remain in the queue. The compacted queue is then sent as the new input queue for the next split kernel invocation and the procedure is repeated until all nodes are divided.

Due to its simplicity and high parallel scalability, the Morton code based algorithm can build hierarchies extremely quickly. It just performs a sorting operation to bucket the primitives to their correspondent nodes, avoiding the evaluation and selection of a split point. Although being the fastest construction method, it does not achieve the best BVHs for collision detection. The reason is that the splitting point is always static and does not adapt to the distribution of the geometry, which could lead to very non-balanced trees causing an increase of work during the collision detection phase.

The SAH construction algorithm performs a more balanced partition of the primitives across the hierarchy. This scenario can be favorable during the BVH traversal by reducing the number of levels covered to detect a collision, as well as the number of elementary tests done in the next phase. As a disadvantage, it has the lack of parallelism between processors in the first splits and the lower resources utilization in very small splits at the last hierarchy levels, plus higher compaction costs. To solve the first issue, the authors present a hybrid algorithm that uses the LBVH method in the initial levels to maintain the parallelism and processes the remaining active splits with the SAH scheme, without compromising the hierarchy quality. In order to deal efficiently with the small splits, the kernel writes those splits to a local work queue (since local memory access is fastest than global memory) and processes them at the last step in one single run, using as few threads as possible (32 in this case) to reduce memory accesses and memory bandwidth.

BVH traversal can also be made in parallel by the GPU to increase collision detection performance. For example, a naïve approach for a BVH traversal is to assign a limited number of leaf nodes in both trees for each thread to test, in order to distribute the work in equal parts. Each thread would be responsible for traversing both BVHs (or one BVH built with pairs of BVs to test from each object), starting on roots and descending only through the branches that would lead to the assigned leaf nodes, as well as doing elementary tests between primitives in case of potential collisions. With this implementation, threads are completely independent from each other, allowing them to execute without waiting for others to finish (to obtain results) and does not use shared memory, avoiding synchronization problems.

However, despite these advantages, this solution does not take maximum benefit from the large number of threads available. The main problem is that collisions in a cloth mesh can occur in localized regions, producing high variance in the number of overlap and collision tests that each thread has to do. In other words, some threads can process branches with few or no intersections at all and terminate their work and yet, need to wait for other busy threads to finish, resulting in many idle threads and less productive work. Furthermore, higher level nodes are visited and tested repeatedly by various threads, which is clearly unnecessary. To prevent these problems, some techniques were introduced, mostly based on work queues, for a better task distribution.

The implementation presented in [21] uses one work queue per core (shared by several threads) to store pairs of BVs to test for overlap. Each work queue is initialized with one pair from the front (if the number of threads are larger than front size, there are threads without initial nodes), making each core traversing a different branch and avoiding repeated tests in the same nodes. At each iteration, a thread dequeues one node from its shared queue (queues are processed in parallel using atomic operations) and checks it for intersection. If the two BVs overlap, two new nodes are generated (or four nodes, depending if we choose to descend from one or both nodes of the pair) and enqueued again. Whenever a work queue becomes full or empty, making impossible further work, a global counter, visible by all threads, holding the number of idle cores, is atomically incremented and the kernel execution is stopped in that core.

Every thread reads that counter after processing a node pair and checks if its value already exceeded a user-defined threshold (50% of idle cores is a good value). If yes, the work queue is written to global memory and the kernel is aborted. After all cores aborted or finished their executions, a kernel is launched to examine all work queues and assign the same number of tasks to each one. The algorithm is repeated until every possible colliding pair is processed.

However, the algorithm performance depends on front size. A small front does not allow for great parallelism by leaving too many threads without work, resulting in a modest speed-up when compared to a sequential approach.

A similar implementation is presented in [15] using both multi-core CPU and GPU architectures to deal with CCD.

A BVH containing pairs of BVs to test (from two objects) is traversed through a CPU parallel algorithm. Each thread has its own queue, initialized with one pair from the BVH front. This pair and its sub-tree are traversed to find possible collisions. Every time a node is tested positive, its child nodes are enqueued for future examination and the node is updated (lazy update). When a queue gets empty, it is necessary to do a task reassignment, inserting new nodes to be processed by the idle thread. This is done as follows: a thread with an empty queue requests a node by signalling a scheduling queue from a busy thread (it chooses the thread with more primitives in its first pair of the queue). As soon as the busy thread checks its scheduling queue and finds a request, sends its first node to the idle thread and continues to process its nodes. This is done again and again until all nodes are scanned. All pairs of probable colliding primitives are putted in a queue to be tested on GPU. Note that this approach could be also implemented on GPU with few modifications.

On a second phase, a single CPU thread sends segments containing thousands of triangle pairs to the GPU. The collision tests between triangles are then performed in parallel by the device and results are sent back to the CPU.

2.3 OpenCL framework

OpenCL [23, 9, 5] is a framework for parallel programming that allows developers to write efficient code in a C based language capable of running on devices like GPUs. It is similar to CUDA, but unlike this, OpenCL is not restricted to NVIDIA's GPUs, providing a standard solution. Moreover, OpenCL is device agnostic, which means that its utilization is not confined to GPUs but to every device that meets OpenCL requirements.

An OpenCL application runs on a host (usually the CPU) that is connected to one or more devices. A device is divided into one or more compute units (in a GPU it corresponds to a set of multiprocessors called Streaming Multiprocessor (SM)) and each compute unit can have several processing elements (equivalent to the multiple cores within a multiprocessor, each one called Streaming Processor (SP)).

The host executes a program that manages the functions (kernels) that run on devices. It allocates memory space on the device and transmits the data required to kernel execution. Each instance of a kernel is called a work-item (equivalent to a thread) and each work-item runs the same kernel code. Work-items are grouped into work-groups, providing a more coarse-grained organization. Each work-group is executed by one compute unit and all its work-items execute concurrently.

Work-groups initially are distributed to the available compute units (multiprocessors) and as they finish, new work-groups are assigned to the idle multiprocessors. The threads within a multiprocessor are executed in groups of 32, named *warps*. Those 32 threads run one common instruction at a time. Work-groups are organized in an N-dimensional grid (where N is one, two or three) that limits the total index space available (Figure 2.6) and have IDs to identify their position inside the dimensional space. Likewise, work-items have local IDs within their work-group and global IDs along the whole space. These identifiers can be very useful, for example to assign to each thread a different position on an array, enabling parallel access. Work-items can access data from different memory locations in the device, as we can see on Figure 2.7.

- Global memory: each thread can read or write from global memory, but only the host can allocate space. It is shared by all work-items from all work-groups.

- Constant memory: it is initialized on the host and remains constant during kernel execution, which means that is a read only memory.

- Local memory: each work-group has a local memory, shared by all their threads, providing faster access than global memory.

- Private memory: it's a private memory location to each work-item, but significantly smaller.

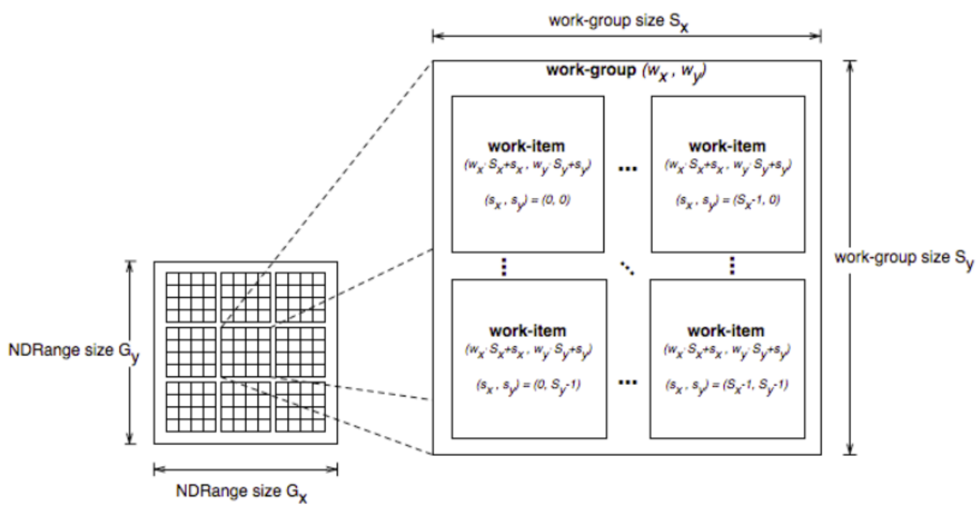

work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

Figure 2.6 Index space orgainzation on OpenCL: example of work-item and work-groups within a two dimensional index space

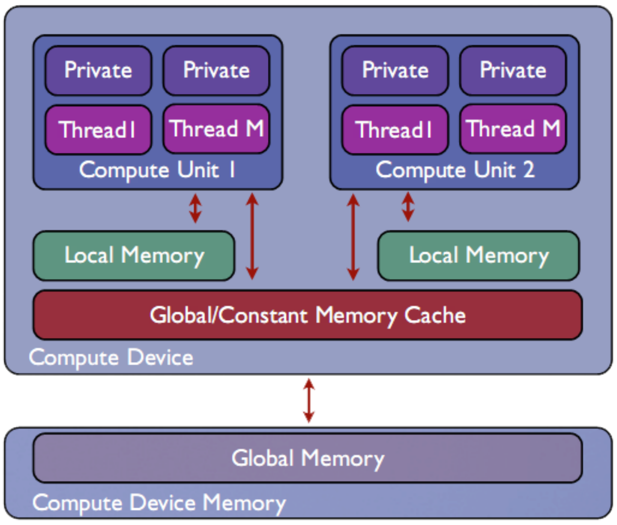


Figure 2.7 Memory architecture in OpenCL device: private, local, constant and global memory locations

During kernel execution, there is no guarantee of memory consistency. In other words, some work-item can access a memory location and read a value that is already outdated. However, it's possible to explicitly enforce memory consistency in some point during the execution, by invoking a thread synchronization. Calling a barrier instruction guarantees consistency in local and global memory within the same work-group.

In conclusion, OpenCL provides a flexible way to take advantage of GPU resources and distribute the computational work efficiently across several cores. This parallelization should be very useful in the cloth collision detection context, since the meshed objects are usually composed by large amounts of primitives, which can result in a high number of collision tests. By using the GPU, it is possible to distribute each primitive or set of primitives within an object across several threads during a BVH construction, BVH update or narrow phase collision detection, while in the BVH traversal we can perform BV overlap tests in parallel. Besides the high parallelism, it is also possible to execute operations in local memory, reducing the memory access timings.

These approaches are shown in the next chapter where we present our OpenCL implementation for cloth simulation collision detection detection.

3. OpenCL Kernels

3.1 Introduction

In this chapter we present an implementation, based on OpenCL kernels, to handle collision detection on the GPU. This implementation deals with discrete collision detection between a piece of cloth and other mesh objects. However, we did not focus on self collisions in the deformable model neither on the collision response mechanism. This GPU kernels were integrated in the cloth simulator developed in [6], which allowed us to have a basis for comparison between CPU and GPU performances.

The kernels developed include the construction of a BVH for each simulation object, two different approaches to traverse the hierarchies and find the colliding primitives, a method to perform parallel elementary tests between primitives and a BVH update scheme.

In the following sections we will explain in detail all of these kernels and our implementation choices.

3.2 Bounding Volume Hierarchy Construction

3.2.1 BVH Construction Kernel

As we have seen in the previous chapter, an option to efficiently detect collisions between meshed objects is to map them into a BVH. We have chosen to use binary trees in our BVH implementation because they are simpler and faster to build. Specifically, for each tree level we only need to perform one split per node, generating two different subsets of primitives.

Concerning the hierarchy construction, the process is composed by three main phases, for each node: choosing the split point within the BV; reordering the primitives either for the left or right sides of the split point; computing the two new child BVs (in a binary tree) from the reordered primitives.

However, to build the root node we only apply the last step, since the root is built from the entire set of primitives while the remaining nodes are defined from their ancestors' primitives. To simplify, we have different kernels for the root and for the rest of the hierarchy nodes. In both construction kernels, each BVH node is built by one work group to ensure that no synchronization between different groups is required.

3.2.1.1 Root Construction Kernel

To construct the root node, each thread from the assigned group reads a primitive from global memory to a private register, in order to create a bounding volume from their vertices' coordinates, as shown in Step 1 of Figure 3.1 and detailed in Appendix A.3.1.

The nodes' BV construction can be viewed as a computation of maximum and minimum coordinate values of the vertices from the primitives along several directions. For example, to calculate the two BV planes parallel to x axis, we need to find the smaller and the higher vertices' values along that axis. To compute the threshold values in the kernel, each thread writes into a local array (with size equal to group size) the minimum value of its primitive's vertices along the chosen direction (Step 2). For instance, if the primitive is a triangle and the direction is the x axis, we check the tree vertices' coordinates and choose the smallest value of x. After the array is completely filled, the entire group performs a reduction operation [13] to obtain the minimum value and updates one BV face with it, as indicated in Step 3 of Figure 3.1.

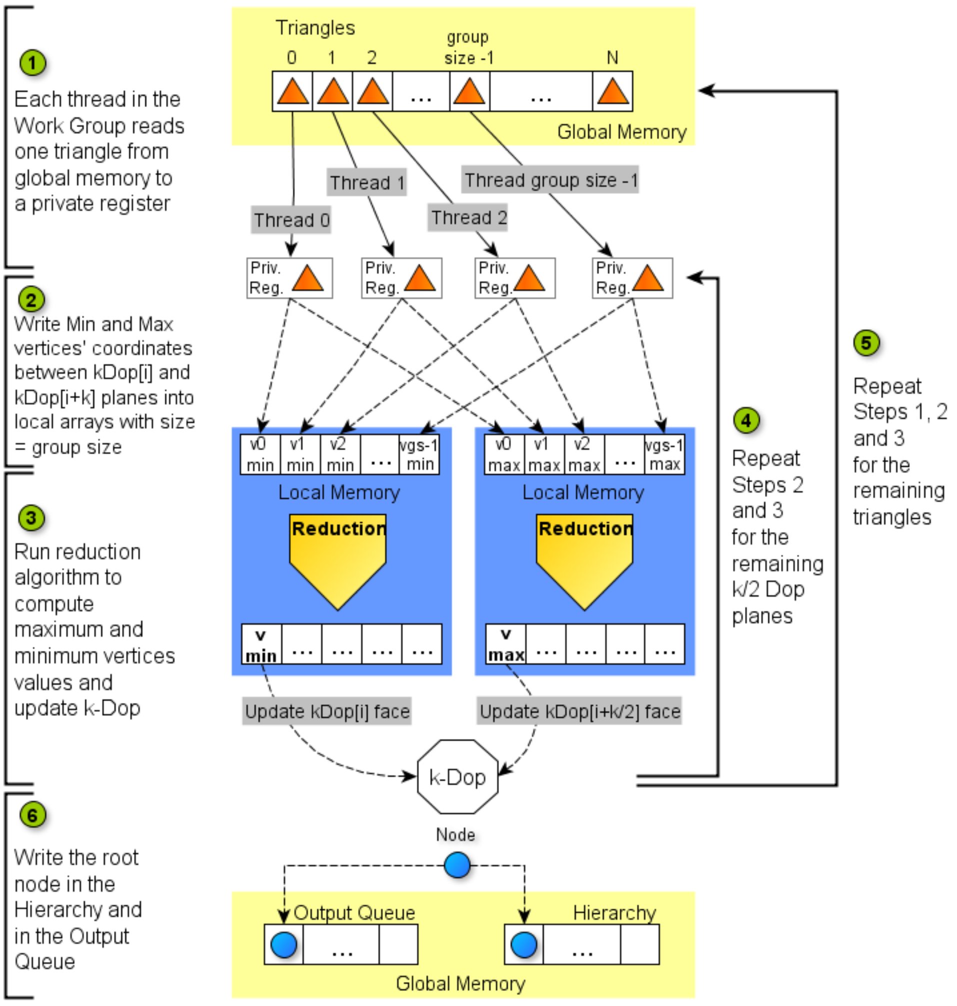

1
Each thread in the Work Group reads one triangle from global memory to a private register
2
Write Min and Max vertices' coordinates between kDop[i] and kDop[i+k] planes into local arrays with size = group size
3
Run reduction algorithm to compute maximum and minimum vertices values and update k-Dop
6
Write the root node in the Hierarchy and in the Output Queue
Triangles
0
1
2
group size -1
N
Global Memory
Thread 0
Thread 1
Thread 2
Thread group size -1
Priv. Reg.
v0 min
v1 min
v2 min
vgs-1 min
v0 max
v1 max
v2 max
vgs-1 max
Local Memory
Reduction
v min
v max
Update kDop[i] face
Update kDop[i+k/2] face
k-Dop
Node
Output Queue
Hierarchy
Global Memory
4
Repeat Steps 2 and 3 for the remaining k/2 Dop planes
5
Repeat Steps 1, 2 and 3 for the remaining triangles

Figure 3.1 Steps to build the root node from a set of triangles

We use a 18-Dop as BV and the faces are stored in 18 length array where the minimum and maximum values in a given direction are stored in positions i and $i+k/2$ respectively, with k representing the number of Dop faces (for instance, over the x axis the threshold values are stored in positions 0 and 9). The process is then repeated for the maximum value in the same direction and spread to all the remaining directions needed to compute the BV faces (Step 4). In our case, to define all the faces of a 18-Dop volume, the reduction is executed over x, y, z, $x+y$, $x+z$, $y+z$, $x-y$, $x-z$ and $y-z$ directions. If the number of primitives is larger than group size, we repeat the above steps for the remaining primitives, processing them in blocks with the group size (Step 5).

When all the BV values are computed, one thread is responsible to add some information fields to the node, such as parent, left child and right child indices, node's level in the hierarchy or start and end indices of the primitives within the volume. The new node is then written to the hierarchy array and also to a work queue to be processed during the next kernel invocation (Step 6 in Figure 3.1).

In the root construction kernel we also have to initialize an array with the primitives' indices. At the beginning, the indices in the array correspond to the primitives' order in their array. However, during the construction process we have to reorder primitives' indices to set them in the correct BVs. Instead of change the primitives array order, we switch their indices in the primitives indices array. The reason why we do this, has to do with the fact that the primitives array on the GPU is not synchronized with the CPU version. Once the triangles are created on the CPU, they maintain their positions in the array during the entire simulation and we don't change their indices to match modifications done in the kernels. Every time the primitives are updated (acceleration, velocity, position, etc) on the CPU, we transfer them to the GPU and access them through the primitives indices array. This way it is possible to utilize the input data and read the correct primitives according to the BVH organization. Unfortunately, this represents an extra step each time we need to access a primitive because we first have to load its index and then the corresponding primitive. However, the initialization process is easily done in parallel by all the threads in which each thread writes its identifier (global id) in the same index (value 0 in index 0, value 1 in index 1, etc).

3.2.1.2 Remaining Hierarchy Construction Kernel

After having the root node built and the indices array initialized, the remaining nodes can be constructed from their ancestors. The task is accomplished on a level by level basis. The following construction kernel invocations have as input a queue created in the previous level with all the non leaf nodes (Figure 3.2 and Appendix A.3.2).

Each input node is read by a different work group which will subsequently split it in two child nodes. Therefore, one thread within the group loads the node from global to local memory and then the remaining threads read it to their private registers (Step 1 in Figure 3.2). Once all threads have their copy of the node, the process of splitting and creating new nodes can start. To choose the best plane to divide the node we use a method presented in [17] that proved to be the fastest among other possibilities: pick the plane where there is the longest distance between two opposite k-Dop faces. This is achieved by computing the difference between k-Dop[i] and k-Dop[$i+k/2$] planes for values of i from 0 to $k/2$.

Then we need to select a good position for the splitting plane. One possibility is to split by the median, in other words, sort the primitives based on their centroids and assign the left half to the left node and the right half to the right node. The Bitonic Sort [16] algorithm is an option to order the primitives that performs well on small sized arrays (capable of being sorted inside a single group) but it is sub efficient for large sequences when compared with the Radix Sort [13, 26]. Although radix sort is particularly suited for integer keys (specially with a small number of bits), it is also possible to sort floating point values by converting them into sortable unsigned integers [14, 27] through a very inexpensive operation. By using this algorithm, dividing the primitives by the median may be an efficient splitting method. However, results in [17] showed that splitting by the mean always produces a hierarchy with smaller total volume than using the median, despite generating less balanced hierarchies.

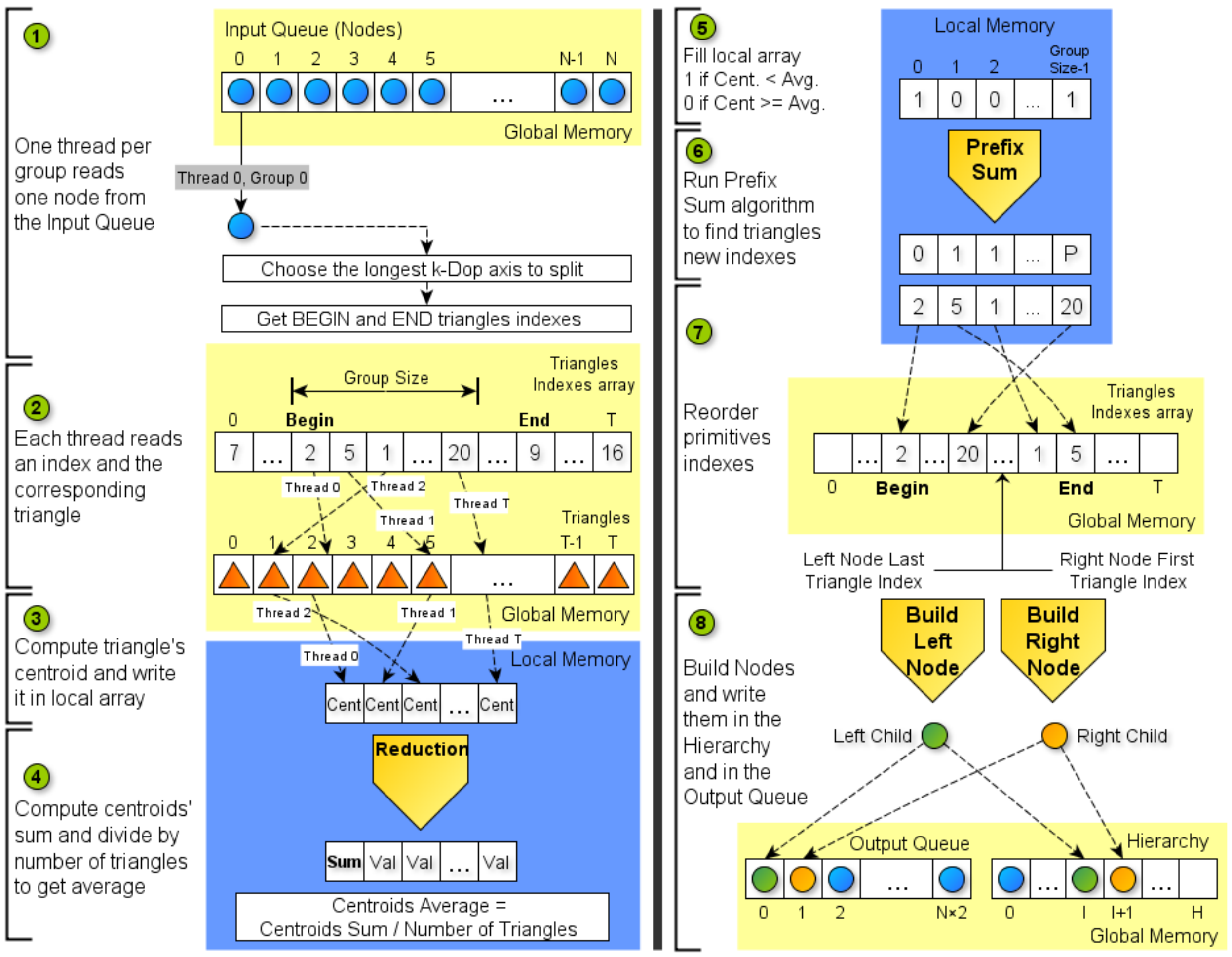

1
One thread per group reads one node from the Input Queue
Input Queue (Nodes)
Global Memory
Thread 0, Group 0
Choose the longest k-Dop axis to split
Get BEGIN and END triangles indexes
2
Each thread reads an index and the corresponding triangle
Group Size
Triangles Indexes array
Begin
End
Triangles
3
Compute triangle's centroid and write it in local array
Local Memory
Reduction
4
Compute centroids' sum and divide by number of triangles to get average
Centroids Average = Centroids Sum / Number of Triangles
5
Fill local array 1 if Cent. < Avg. 0 if Cent >= Avg.
Prefix Sum
6
Run Prefix Sum algorithm to find triangles new indexes
7
Reorder primitives indexes
Left Node Last Triangle Index
Right Node First Triangle Index
8
Build Nodes and write them in the Hierarchy and in the Output Queue
Build Left Node
Build Right Node
Left Child
Right Child
Output Queue
Hierarchy

Figure 3.2 Steps to build two child nodes from their ancestor

Based on that, the decision was to use the mean method and compute the average of primitives' centroids in the chosen plane through a reduction algorithm and sort them into either the left or right sides using a scan algorithm. To perform that task in parallel each thread reads one triangle index (in the range between the node's primitives' indices) and loads its subsequent triangle (Step 2). Then extracts its centroid in the chosen plane (given by the average between all primitive vertices' coordinates in that plane) and writes it down in a local memory array, as illustrated in Step 3. The work group runs a reduction to find the centroids' sum and divides it by the number of primitives to get the average point (Step 4).

After that, it is necessary to reorder primitives' indices based on the split point. If the triangle's centroid value is lower than the split point, a 1 is written into a local array, otherwise it's filled with a 0 (Step 5). This array is used to perform a prefix sum and from the values obtained it's possible to calculate the new positions for primitives' indices (Step 6). Left sided indices are placed in consecutive positions from left to right, starting from input node begin index, and the same applies to right sided indices, in the opposite way, starting from end index, as showed in Step 7 of Figure 3.2. If the number of node's primitives is larger than the group size, we need to keep processing indices in blocks with group size, starting again from Step 2.

At the end, we must have all the left and right new nodes' indices ordered between the input node bounds and in addiction know the index where one ends and another begins. The next stage (Step 8) consists then in constructing both left and right BVs with their newly primitives through the same mechanism used in root building (Figure 3.1), with the difference that the left child is built with primitives in the range of *begin* and $last\ left\ node\ index$ and the right child with the primitives in the range $last\ left\ node\ index + 1$ to end.

Hereafter, one thread sets in both nodes their parent index as the input node index, as well as their left and right child indices. If the node has less primitives than the threshold defined for a leaf, its children indices are set with -1 value, signaling that it is a leaf node. In addiction, its *begin* and *end* fields, pointing to the primitives indices array, are also saved to be used in the next split. They are finally written in the hierarchy and in the output queue to be processed in the next kernel invocation (Step 8).

3.2.2 BVH Construction Kernel Optimizations

The above construction kernel has two main bottlenecks, as evidenced in [7]. One of them is the large number of nodes with few primitives in the lower levels of the hierarchy. The second bottleneck happens in the initial hierarchy nodes, due to their large size and the lack of parallelism by assigning only one work group per node.

These drawbacks are evidenced in Figure 3.3, where the constructions timings per level for an object with approximately 1 million primitives (Happy Buddha model, illustrated in Figure 4.1) are shown. As we can see, the first levels of the hierarchy are considerably slow when compared with the intermediate levels (the root level construction is faster than the second level, since we do not need to find a split point, neither to reorder the primitives), while in the last levels we notice a decrease in the performance related with the queue size.

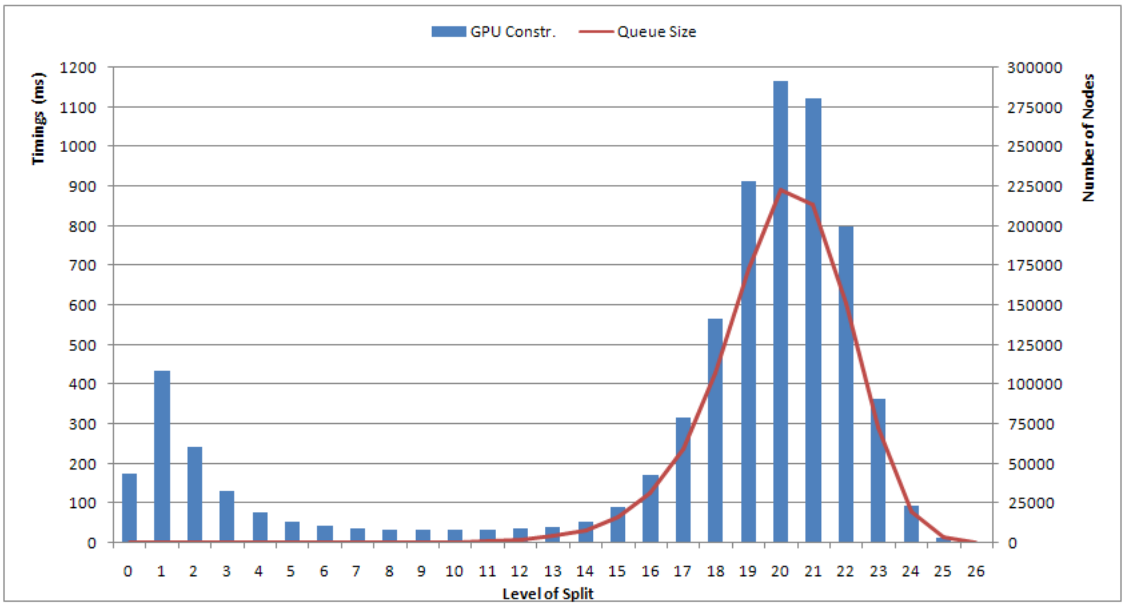


Figure 3.3 Construction timings per level for for the Happy Buddha model: construction is slow in the first levels due to the nodes' size, while in the last levels the performance decreases due to the queue size.

In the following subsections we introduce three measures to reduce these performance issues.

3.2.2.1 Adaptive Work Group Size Optimization

In our construction kernel, each node is split by an entire work group. As we build new hierarchy levels, the amount of primitives per node decreases. When the number of primitives in the node is less than the group size we can have many idle threads (or threads doing unnecessary work) revealing a poor utilization of work group parallelism, that is reflected in a decrease of performance.

To solve that problem we need to adapt the work group size to the number of primitives per node, in order to have the maximum use of all threads. Our solution was to estimate the average node size at each level and reduce the group size to follow that value. The number of threads within each work group should be always higher than the node size and also a power of two, since it is a necessary condition to execute the scan and reduction algorithms. We compute the average node size at each level by dividing the total number of primitives by $2^{level-1}$. Whenever that value is above the work group size, we round it up to the next power of two and set it as the new work group size.

However, if we have less than 32 threads per work group the multiprocessors are not running a full warp, which means that we are not taking advantage of all their capabilities. To split very small nodes we introduce a refined approach explained in the next section.

3.2.2.2 Small Splits Optimization

To split very small nodes we use an algorithm based on the idea presented in [7] to deal with small splits. The idea is that each small node is processed by a single thread using its private memory registers, avoiding the local memory accesses and the synchronization needed to split larger nodes across all group threads (kernel detailed in Appendix A.3.3).

In the construction kernel we define a threshold value for the maximum number of primitives contained in a small node. During the kernel execution all the nodes above that threshold are filtered into a local array instead of being split by the current work group. Once all nodes from the input queue are processed, each thread within the group loads and splits individually one or more small nodes from the local array. By individually we mean that all the steps needed to split the node and generate its two child nodes are made sequentially by only one thread. With this method there are several threads doing work in parallel and less idle threads. Nevertheless, the threshold value should be smaller than the group size in order to compensate the increase of work per thread when compared to a split executed by the entire work group. In the end of the kernel, one thread per group atomically increments a global counter with the number of small splits done in that group, to account the total of small nodes in that level.

Before the next kernel invocation, we compare the number of small nodes with the queue size: if the small nodes exceed a percentage limit (another defined threshold) in the queue, we invoke a different kernel where each thread begins immediately by reading the nodes from the input queue in parallel, avoiding the filtering process (conducted by only one thread per group in the construction kernel). In other words, each node will be processed by a single thread instead of being processed by an entire work group, which results in more work per thread, less idle threads and no need for synchronization of work group threads during the kernel execution.

This modification in the construction mechanism represents a considerable improvement in terms of performance, as we will see in chapter 4.

3.2.2.3 Large Nodes Optimization

In this section we address the bottleneck problem in the top hierarchy levels by introducing an improved construction mechanism for large nodes. Instead of assigning a single work group per node, we want to use several work groups, cooperating to build those large nodes.

The basic idea is that each group should process a block of node's primitives (or more, depending on the number of work groups assigned) during the building process. The construction process can be divided into three main phases: choose the split point, reorder the primitives and build the two new child nodes. Each one of this phases is done through parallel algorithms such as scan and reduction. Note that for the root node we only have to perform the last phase, since we already have the initial set of primitives ready for the k-Dop construction.

To choose the split point we need to perform a reduction over primitives' centroids. That operation can be done by several groups by assigning a different block to each group (Figure 3.4 and Appendix A.3.4). After reducing the array, each group writes its sum into a block sums array (Steps 1 and 2). Then, we invoke another kernel with one group that reduces the block sums array to find the total sum and divide it by the number of primitives to get the mean point, as illustrated in Figure 3.4 - Step 3 (the loading of the block sums array from global to local memory was ignored in the figure). In alternative, we can use only one kernel: in the first kernel, one thread per group increments a global counter with the total sum through atomic operations.

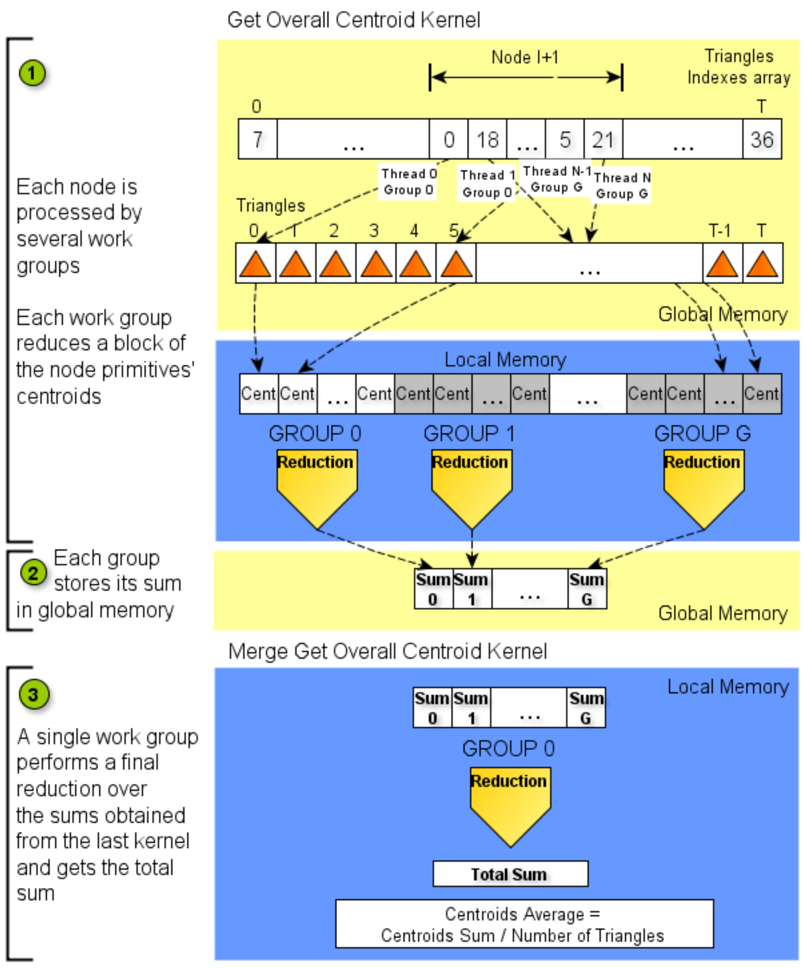

Get Overall Centroid Kernel
1
Each node is processed by several work groups
Each work group reduces a block of the node primitives' centroids
Node I+1
Triangles Indexes array
0
T
7
...
0
18
...
5
21
...
36
Thread 0 Group 0
Thread 1 Group 0
Thread N-1 Group G
Thread N Group G
Triangles
0 1 2 3 4 5
T-1 T
Global Memory
Local Memory
Cent
GROUP 0
GROUP 1
GROUP G
Reduction
2
Each group stores its sum in global memory
Sum 0
Sum 1
Sum G
Global Memory
Merge Get Overall Centroid Kernel
3
A single work group performs a final reduction over the sums obtained from the last kernel and gets the total sum
Local Memory
GROUP 0
Reduction
Total Sum
Centroids Average = Centroids Sum / Number of Triangles

Figure 3.4 Kernels to get the split point for large nodes. The first kernel is executed by several groups. In the second kernel, a single group performs a final reduction to obtain the overall centroid.

To reorder primitives' indices each group runs a scan operation over the primitives' centroids flags (0 if the centroid is in the left of the split point, 1 if it is in the right side) to compute their new indices. Then, the scanned array is saved in global memory (not shown in Figure 3.5)and each group also writes its prefix sum (last position in the array) into an array with all groups prefix sums, as shown in Step 1 of Figure 3.5 (in the figure were ignore the steps that precede the centroids' flags scan, since their basis were explained in section 3.2.1.2). In a second kernel, each group scans the groups sums array to compute its starting offset (global memory loads were omitted in the figure), loads its scanned array block and computes the new indices with that information, either for the left and right side indices, as illustrated in Step 2 of Figure 3.5.

To build the BV each group treats one block of primitives and builds its own BV (Step 1 of Figure 3.6). In a second kernel, one group performs a merge of all the BVs and define the final volume (Step 2 of Figure 3.6).

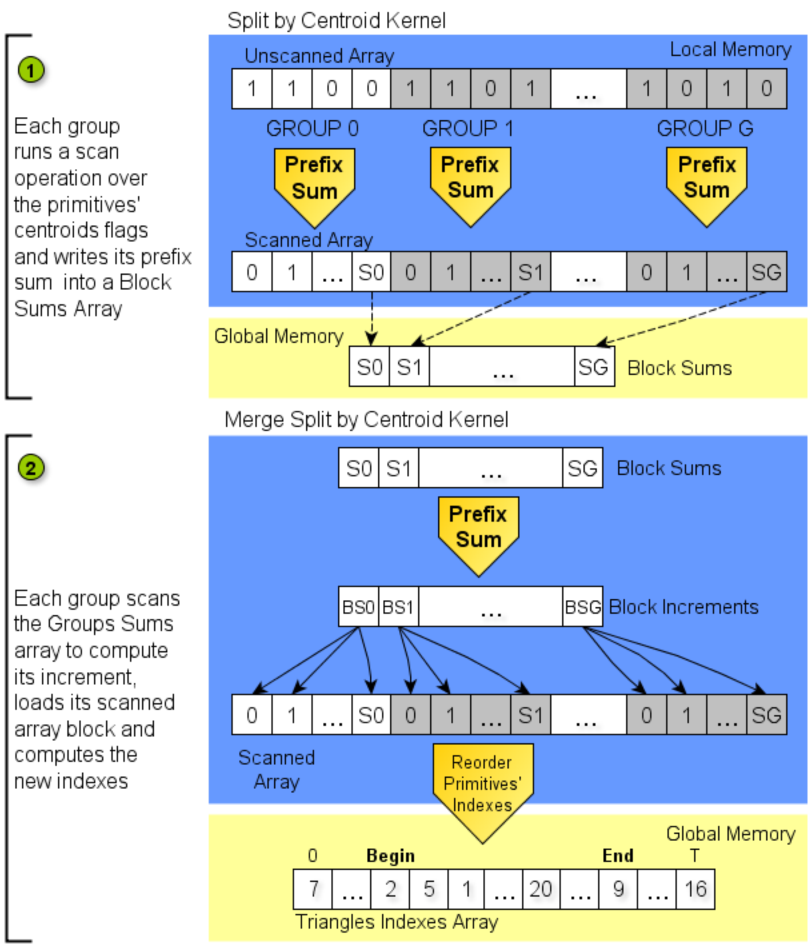

Split by Centroid Kernel
1
Each group runs a scan operation over the primitives' centroids flags and writes its prefix sum into a Block Sums Array
Unscanned Array
Local Memory
GROUP 0
GROUP 1
GROUP G
Prefix Sum
Scanned Array
Global Memory
Block Sums
Merge Split by Centroid Kernel
2
Each group scans the Groups Sums array to compute its increment, loads its scanned array block and computes the new indexes
Block Increments
Scanned Array
Reorder Primitives' Indexes
Begin
End
Triangles Indexes Array

Figure 3.5 Kernels to reorder primitives' indices. In the first kernel each group scans the centroids' flags array and saves its block sum. In the second kernel, each group obtains its block increment by scanning the block sums array and computes the new indices.

In conclusion, to use all the GPU resources we need to divide each one of the three phases (illustrated in figures 3.4, 3.5 and 3.6) in two kernels. In the first we distribute the work by several work groups and in the second we merge the intermediate results to get the final product.

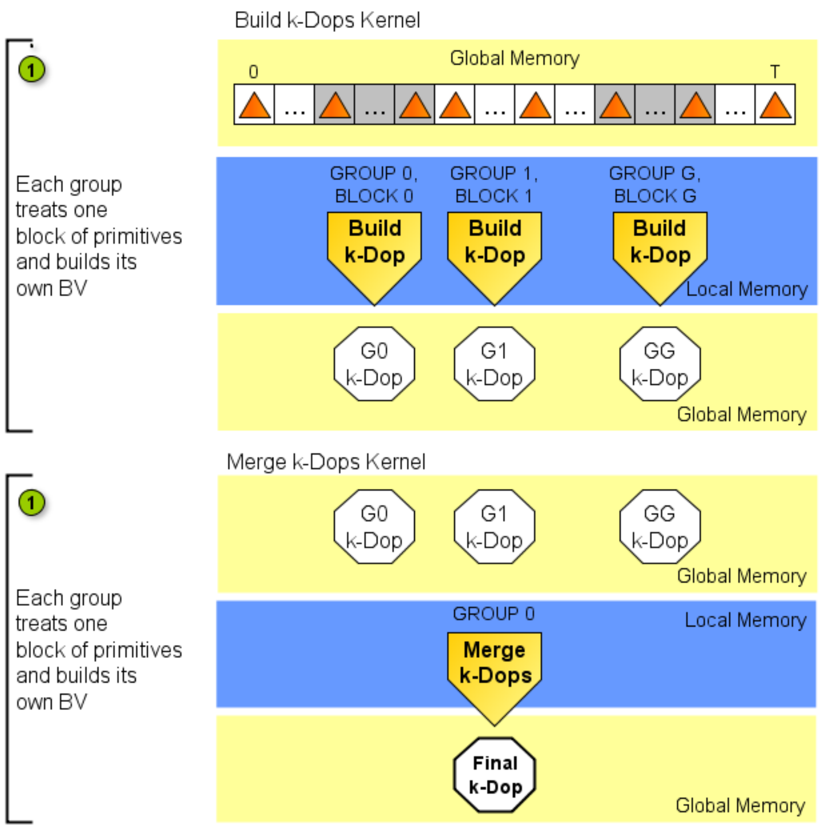


Figure 3.6 Kernels to build a BV from several blocks of primitives. In the first kernel each group builds a BV from its block of primitives. In the second kernel, one work group merges all the BVs constructed in the previous kernel and defines the final BV.

3.2.3 Compaction Kernel

3.2.3.1 Compaction Kernel for Small Arrays

Whenever the construction kernel creates leaf nodes, it is necessary to perform a compaction in the output queue, leaving only the internal nodes to be split in the next construction kernel invocation. For a small queue, capable of being processed by only a single work group at once, we only invoke one work group (Figure 3.7 and Appendix A.3.5.1). In our kernel, each thread processes two nodes, so the queue can have up to twice the size of the work group.

Each thread within the group reads two nodes from the queue, separated by a distance equal to the group size, into their private registers. For example, the first thread will read nodes at indices 0 and groupsize-1, the second at indices 1 and group size and so on. This way, every thread participates in the first iteration of the prefix sum algorithm used to sort the nodes. Then, we write either 0 or 1 into a local array depending if the node is a leaf or not. In the next step,

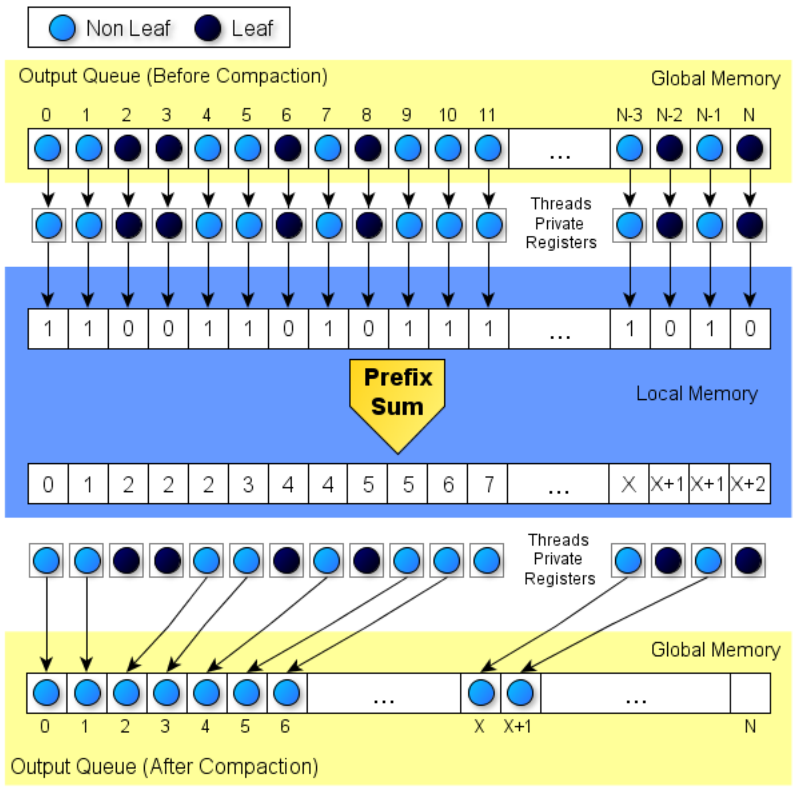

Non Leaf
Leaf
Output Queue (Before Compaction)
Global Memory
0 1 2 3 4 5 6 7 8 9 10 11 N-3 N-2 N-1 N
Threads Private Registers
1 1 0 0 1 1 0 1 0 1 1 1 ... 1 0 1 0
Prefix Sum
Local Memory
0 1 2 2 2 3 4 4 5 5 6 7 ... X X+1 X+1 X+2
Threads Private Registers
Global Memory
0 1 2 3 4 5 6 X X+1 N
Output Queue (After Compaction)

Figure 3.7 Compaction kernel for small arrays, executed by a single work group

we run a parallel scan algorithm (prefix sum) to compute nodes' new indices. With the array scanned, threads write the internal nodes in their new indices, inside the new output queue.

However, besides removing leafs from the work queue, compaction has an additional function of redirecting some nodes' child indices. This comes from the fact that leaf nodes do not have children, which would create empty spaces between the nodes of the next level of the hierarchy.

3.2.3.2 Compaction Kernel for Large Arrays

To scan large arrays we use a different method executed across several work groups simultaneously and divided into three distinct kernels (for more details, please see Appendix A.3.5.2). In the first kernel, the queue is divided into smaller blocks, capable of being processed inside a single group at once. Each group scans its block and writes the total sum (corresponding to the number of non leaf nodes) into an array of block sums in global memory, as illustrated in Steps 1 and 2 of Figure 3.8.

Then we run a second kernel with a single work group to scan the block sums array, generating an array of block increments (Figure 3.9).

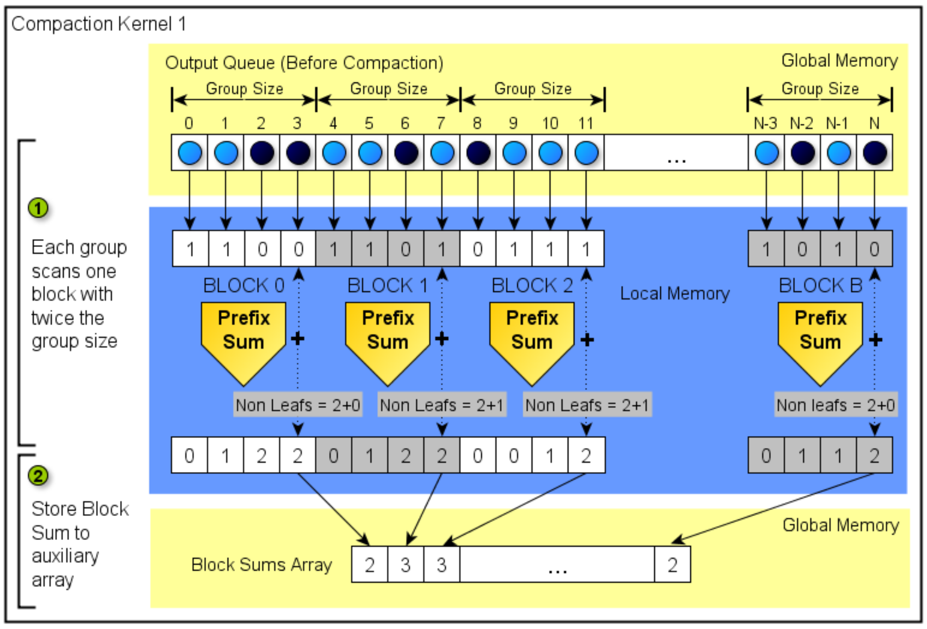


Figure 3.8 Compaction for large arrays: in the first kernel each group scan a block of nodes from the queue, creating a block sums array

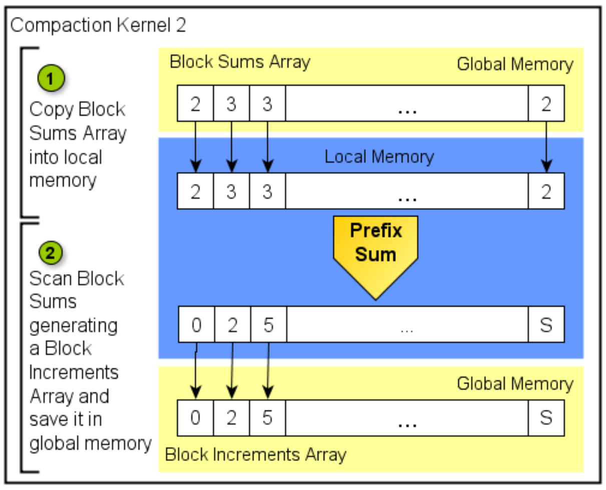


Figure 3.9 Compaction for large arrays: in the second kernel a single work group scans the block sums array, generating an array of block increments

If the arrays is bigger than the group size, the scan is performed several times inside the work group in blocks with group size. For simplification we have only one work group, but the kernel could be also executed by several work groups by assigning each block to a different group. In that case, each work group would have to fill an extra block sums array for the block increments array, that would have to be also scanned in the last kernel. Another option could be computing the block sum scan in the last kernel, without this intermediate kernel. However, this would require that every work group had to perform the same scan operation over the entire array. That situation would be particularly bad if we have more work groups than multiprocessors, meaning that each multiprocessor has to run the same scan operation for multiple groups.

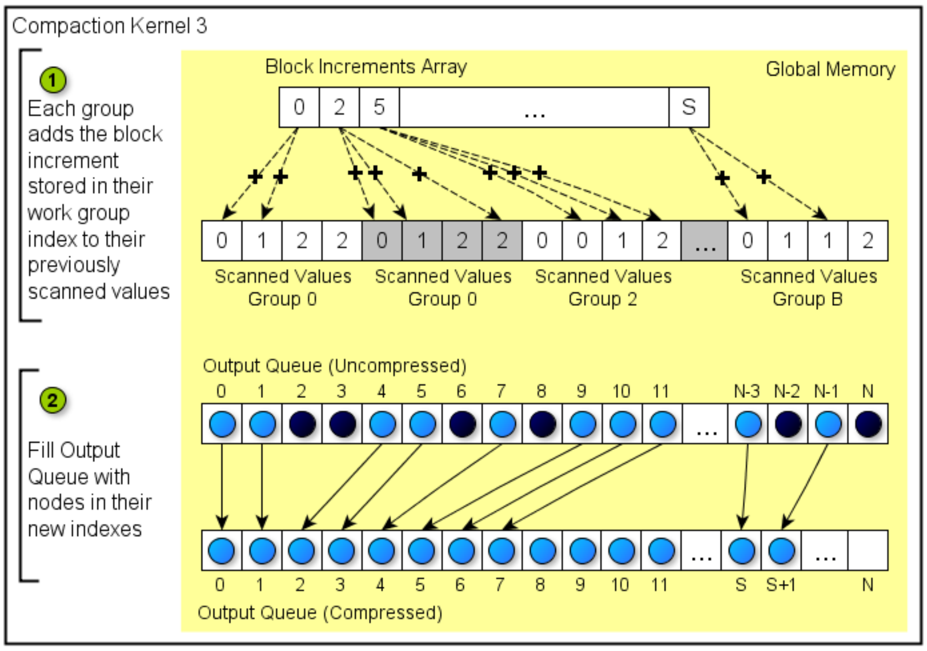

Compaction Kernel 3
1
Each group adds the block increment stored in their work group index to their previously scanned values
Block Increments Array
Global Memory
0 2 5 ... S
Scanned Values Group 0
Scanned Values Group 0
Scanned Values Group 2
Scanned Values Group B
2
Fill Output Queue with nodes in their new indexes
Output Queue (Uncompressed)
Output Queue (Compressed)

Figure 3.10 Compaction for large arrays: in the third kernel the group threads add the block increment stored in their work group index to their previously block scanned values and save their nodes in the new indices

Finally, we run a third kernel in which all the group threads add the block increment stored in their work group index to their previously block scanned values (Step 1 in Figure 3.10). With those values, each thread with a non leaf node enqueues it in the right position and corrects its child indices (Step 2).

3.3 Broadphase Collision Detection Kernel

3.3.1 Simple Broad Phase Collision Detection

In the broad phase we traverse the hierarchies to find nodes whose bounding volumes overlap. As a result we obtain a more restricted list of primitives that have chances to collide, therefore eliminating unnecessary tests between far apart primitives.

The kernel responsible for this operation receives as input the object's hierarchies and returns as output a list with all the potential pairs of colliding primitives. The first step is done sequentially by one thread that reads both root nodes and tests their BVs for overlap (Figure 3.11 and Appendix A.4.1).

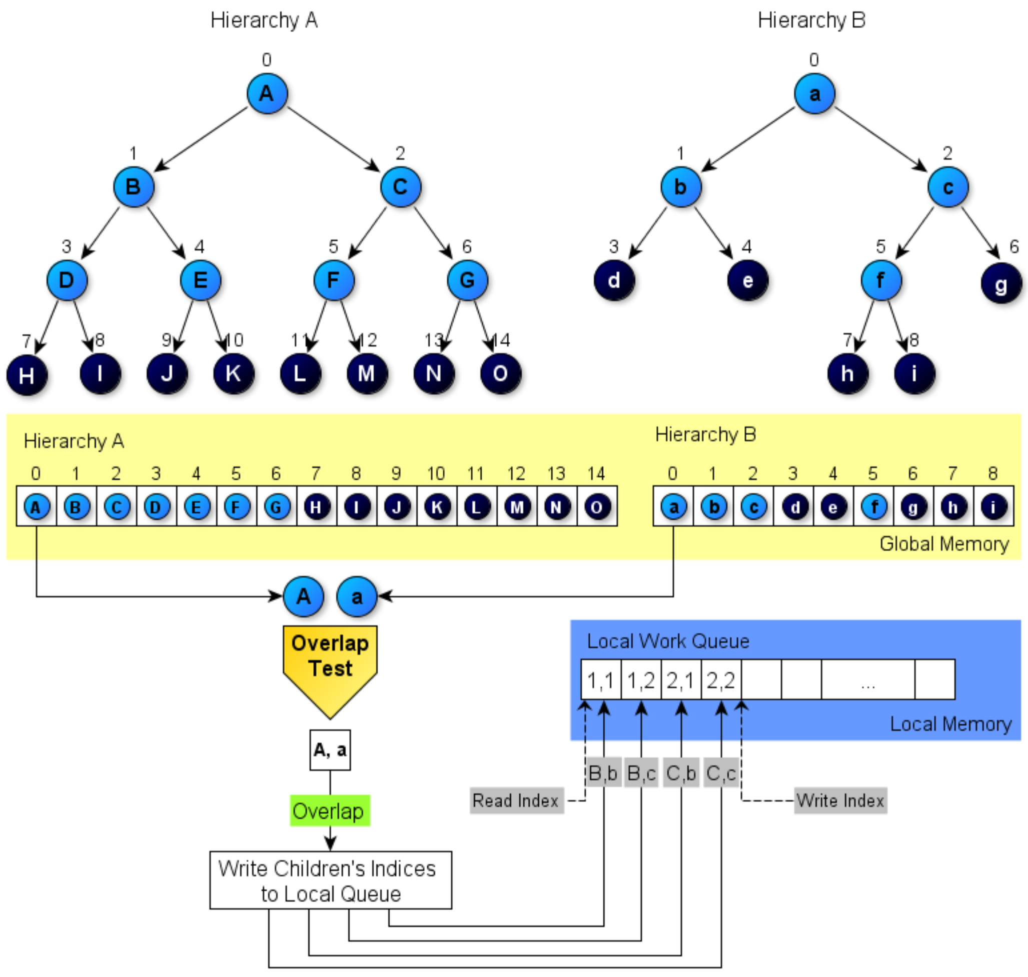


Figure 3.11 Example of hierarchies traversal starting from the roots. When two internal nodes overlap, we write four new node indices into a local queue for further processing.

This overlap check consists in testing if the opposite faces of both BVs intersect, which has a small cost (for k-Dops the operation cost is O(k), where k denotes the number of Dop faces) and is executed by a single thread for each pair of nodes. If the BVs overlap, the thread writes into a local queue all the possible pairings of their children's indices for further processing. In a binary tree it corresponds to four pairs of indices, assuming that both roots are non-leaf nodes. The choice of storing indices instead of directly storing primitives is due to the fact that local memory space is insufficient to keep a large amount of geometry data. After inserting new nodes on local queue a local counter (write index) is also updated, signaling the index of the last enqueued element, in order to avoid reading positions out bounds.

From now on the algorithm can run in parallel by several threads. A thread with an identifier smaller than the difference between read and write pointers can dequeue a pair of indices and load their corresponding nodes from the hierarchies, while increases the read index value. If an overlap exists, the write pointer is incremented by the number of elements added and the indices are placed in the queue. If one node from the overlapping pair is a leaf, we enqueue the indices from the leaf together with the indices from the other node's children, resulting in just two new items (Figure 3.12).

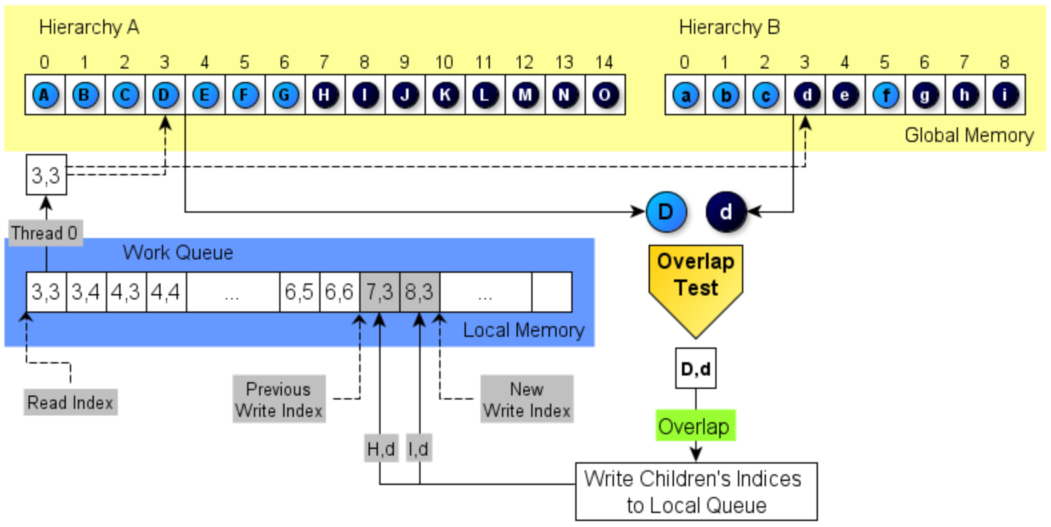


Figure 3.12 Example of hierarchies traversal starting from the roots. When one of the overlapping pair is leaf, it results in two new node indices in the local work queue.

To synchronize this process and prevent different writes on the same position, each thread uses atomic operations to change pointer's values.

In case both nodes are leafs, the thread loads nodes' triangles and stores all the possible pairs of colliding primitives into a global array (Figure 3.13). To ensure that each pair is written in an empty position, it is necessary to atomically increase the array pointer into the last index saved.

The algorithm ends when the read index reaches the same value of the write index, meaning that the processing queue is empty.

This kernel is executed by a single work group to avoid repeat the same overlap tests over different groups. Obviously, this represents a waste of GPU resources, since we have many idle multiprocessors. A smarter approach might be to launch four work groups to test the root nodes and in case of overlap distribute the four children pairing across the groups. Nevertheless, the

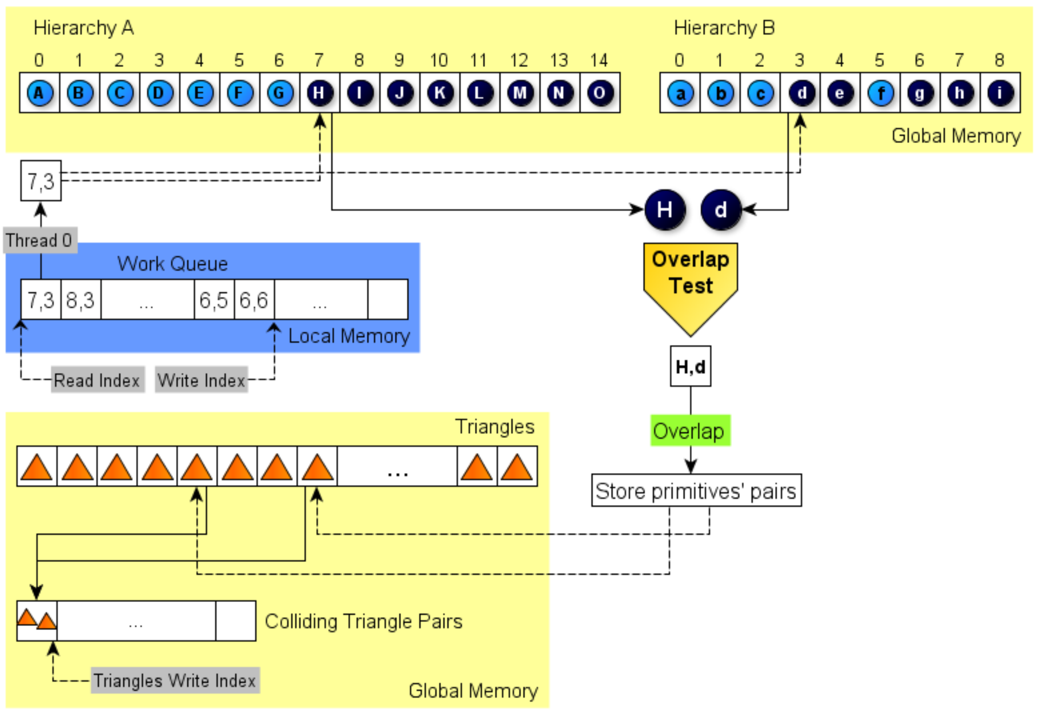


Figure 3.13 Example of hierarchies traversal starting from the roots. If two leafs overlap, we write their pairing primitives into a result array in global memory.

kernel would perform, at most, four times better than the single group method.

To increase the number of initial nodes to test and launch several work groups in parallel we could use a different method, by starting the traversal in lower levels. For instance, we could begin in the first levels with leaf nodes and descend the tree from there. The number of initial overlap tests would be given by the total amount of node pairs from both starting levels of the hierarchies. In an extreme case, this represents testing all the leaf pairs in both hierarchies. It could be a good approach if the majority of the objects' primitives are colliding, when compared to always start from the root nodes but even so, it might result in many unnecessary tests between far away BVs and the algorithm will perform worst when the objects do not collide.

Another strategy could be using the CPU to perform the tests in the first levels of the hierarchy (with fewer nodes), generating a set of colliding pairs of nodes in a sufficient number to distribute among several work groups. By doing this, we were only sending to the GPU, as input, pairs of nodes that really intersect, avoiding unnecessary tests between faraway BVs like in the above method.

However, in order to solve this issue and take maximum advantage of work group parallelism, we use a Front based approach that fully runs on the GPU. In the next section we will explain our implementation.

3.3.2 Front-Based Broad Phase Collision Detection

As explained in section 2.2.2.3, the front concept was introduced to explore temporal coherence during a simulation and reduce the number of nodes traversed in the broad phase. However, in a multi-threaded environment it has another great advantage: allows to distribute the different front elements across several threads (and possibly several work groups) to be processed in parallel. This task division represents a better utilization of GPU resources when compared with the traditional top-down approach, reflected in less work per thread (obvious in the first top levels of the hierarchy). The longer the front, the larger the parallelism achieved and consequent success of this technique. In addition, each node has a distinct sub-tree, enabling each thread to independently traverse the hierarchies and update the front without any type of synchronization.

At the beginning there is only one element in the front, formed by the two hierarchy roots. Traversal process starts in the same way as the non-front version, by introducing elements into a processing queue that are then processed in parallel. The main variation lies on the fact that we need to perform a front maintenance during the traversal in order to provide a valid front for the next simulation step (details can be seen in Appendix A.4.2).

By definition, the front includes all the leaf pairs, as well as all the last non-overlapping pairs tested. In other words, it consists in the set of nodes where the traversal terminates during the collision detection, as we can see in Figure 3.14.

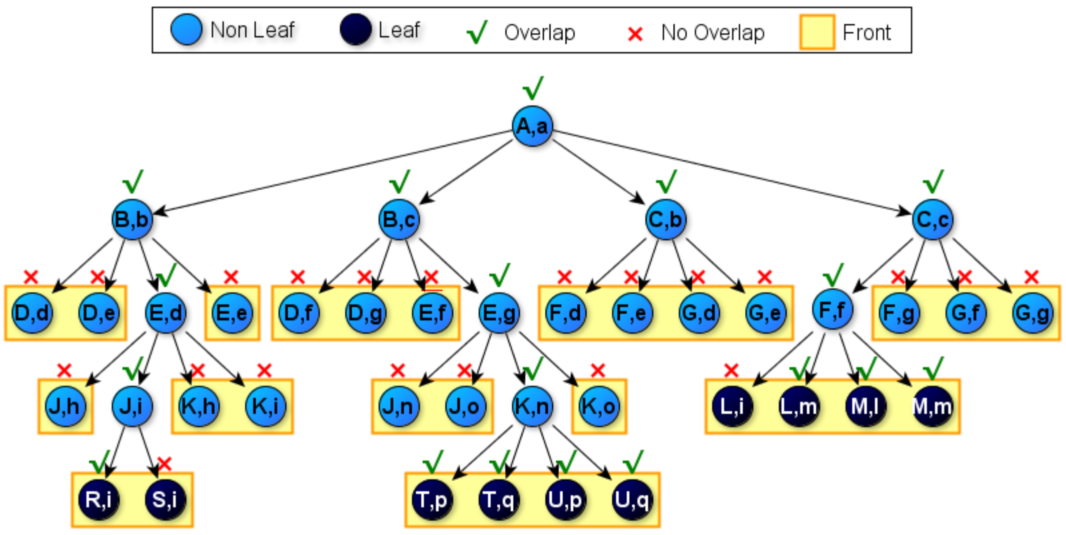


Figure 3.14 Example of front building. The front includes all the leaf pairs, as well as all the last non-overlapping pairs tested

According to this, front update may involve replacing a node by a group of deeper nodes in its sub-tree (lowering and expanding the front) or exchange one or more non-overlapping pairs by one higher element (raising and shrinking the front).

The first operation is done when threads are reading nodes from the queue and checking for collisions (while descending the hierarchies). Whenever a thread reaches a leaf node (colliding or not) or a non-overlapping internal node, the node is added to the front (a global array).

The raising (Figure 3.15) is done after assigning the initial front nodes to different threads. If the initial node pair (R,i) does not intersect, the thread tries to climb the hierarchies. In order to choose the correct node to climb, it needs to look at nodes' (within the pair) level in their hierarchies: if they are in the same level, it is loaded the pair formed by both their parents; if levels differ, we go up by the deepest node. If this upper pair (J,i) does not overlap, the thread continues the ascension in the hierarchy. In other hand, when it represents an intersection, the front is updated with the current pair (R,i), matching the deepest non-overlapping node pair.

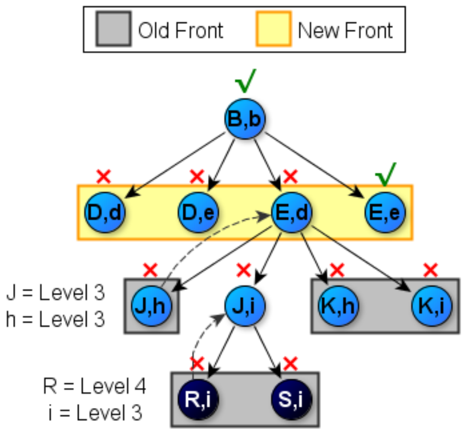


Figure 3.15 Front raising

During climbing procedure the same node can be reached by different threads with different initial nodes. To avoid adding that node multiple times to the front, only the leftmost child is allowed to climb the trees and consequently update the front (in the example figure, only the thread with J,h node climbs the hierarchy).

After the tree traversal is finished, the new front nodes as well as the possible colliding pairs of nodes are written in global memory. If we use several work groups to perform the Broad Phase, each one has a defined space in the array where it can write the nodes. For example, if we define an offset of 1000 nodes, group 0 can write from index 0 to 999, group 1 from index 1000 to 1999 and so on. In that case we need to perform a compaction in both front and colliding nodes arrays to get a valid input for the next kernels. In order to know how many nodes are written by each group, we store the last written index (for both front and colliding nodes) for every work group. Then, we can invoke a compaction kernel with only one group to read the front nodes and the colliding nodes from the old arrays and write them in their new indices in a new array.

However, both our implementations have a drawback: the nodes are read and written in local memory (in a local work queue), which reduces the capacity of the queue, due to the limited local memory size. In our case, when the queue becomes full, threads can't write on it and the new nodes are lost. Only when more nodes are dequeued it is possible to save new pairs of colliding nodes to process. This problem causes a decrease in the collision detection accuracy, since there are branches in the hierarchies that are not traversed and the final number of primitives to test will not be correct. We did not focus in this issue, so our implementation is restricted to a certain number of nodes in the work queue (around 1000 pairs of nodes in our GPU), but there are ways to solve this problem.

A simple way is to have an extra working queue in global memory and use it every time the local queue is full. As soon as the local queue has space, the nodes are written again in local memory.

Another approach is the one previously explained in section 2.2.3. Every time a local queue becomes full (in one work group), all the work groups write their queues into global memory and the kernel is stopped. Then, we invoke the kernel again and redistribute the nodes saved in global memory by the several work groups and resume the traversal process.

3.4 Narrow Phase Collision Detection

In the narrow phase we test the primitive pairs resulting from the broad phase to find which really collide. This process is highly parallel and don't need any synchronization, since each thread tests one pair of primitives independently. To perform the intersection test between two triangles we use the open source algorithm presented by Möller in [22]. Moller's method basically checks the co-planarity of two triangles to find out if they intersect.

The kernel receives as input an array with the pairs of possible colliding triangles and return an array with the triangle indices that really have collided (Appendix A.5). That is our only output, since the collision response was outside the scope of our work.

3.5 Update Kernel

Updating the hierarchy is a way to prepare it for the next simulation step without perform an entire rebuild operation. In BVH building each volume is computed using primitives' coordinates, whereas in update method only leafs need primitive checking and the remaining volumes are refitted by merging their children's volumes, which is less expensive than testing all the primitives. The update kernel (Figure 3.16 and Appendix A.6) uses a bottom-up approach and is invoked once per level.

Since the hierarchy is stored in an array arranged by the levels, we can easily perform the update by sending an extra information (saved during execution of the construction kernel) about the first index of the current level and how many nodes it has. The algorithm starts by computing the number of nodes per work group, in order to distribute the work equally. At each iteration, each thread within a group reads two siblings. This way, after refitting these two nodes it is possible to load and update their parent without any synchronization with other threads. By assigning each node to only one thread, the level is covered very quickly and threads can efficiently update a node because it is not expected an high number of primitives in leafs and merge operation has a low cost. The kernel is invoked once per level and the changes are propagated towards the root.

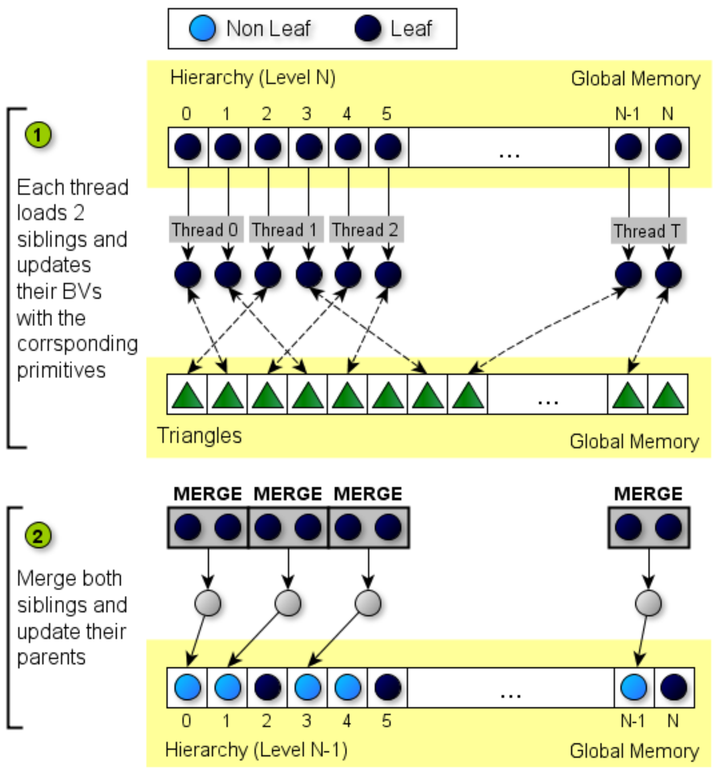


Figure 3.16 Steps to update the hierarchy using a bottom-up approach.

4 . Results and Analysis

We now expose the results achieved with our kernel implementations on the GPU and compare them with the performances obtained from the analogous CPU sequential implementation in the cloth simulator developed in [6]. Our experiments were run on a Intel Core 2 Duo T8300 at 2.13GHz with a NVIDIA Quadro FX 3800. We also have run the more optimized version of the construction kernel with a NVIDIA Tesla C2050 for comparison. The specifications from the above GPUs are described in Table 4.1. All the timings obtained from the GPU are only relative to the OpenCL kernels execution and do not include any CPU-GPU data transfer.

Table 4.1 Hardware specifications of the devices used to run our tests.

Specifications	**NVIDIA Quadro FX 3800**	**NVIDIA Tesla C2050**
Max. Clock Frequency	1204 MHz	1147 MHz
Multiprocessors / Compute Units	24	14
Cores	192	448
Memory Bandwidth	51,2 GB/sec	144 GB/sec
Memory Interface	256-bit	384-bit
Memory Speed	800 MHz	1,5 GHz
Global Memory Size	1 GB	3 GB
Local Memory Size	16 Kb	48 Kb
Max. Work Group Size	512	1024

4.1 Construction Kernels Results

In this section we will show the performance results achieved with the BVH construction kernels. Our results will reflect the optimizations made from the first version of the kernel to the last and optimal solution. To test our implementations, we have used the object models illustrated in Figure 4.1. The construction kernels were launched with a group size of 128 and with a global size of 24×128 in order to give one work group to each compute unit (with the Quadro FX 3880). The compaction kernels have a group size of 256 and a global size of a global size of 24×256, except the local compaction kernel where we run a single work group. Despite the maximum group size of the Quadro FX 3800 being 512, we cannot have that many threads running because they exceed the maximum number of the device private registers (specially on more complex kernels with lots of private variables). Since we have only one work group per compute unit and in some cases one work group is responsible for processing more than one node, the scheduling is done inside the kernel (and not by the hardware scheduler) by computing for each group the memory positions that they can access (array indexes).

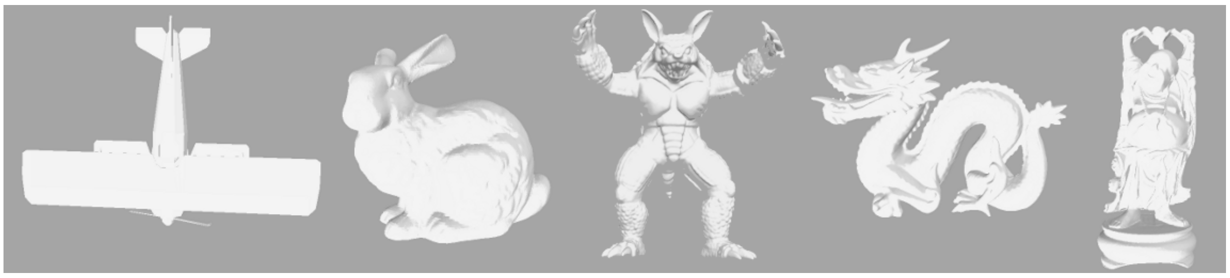


Figure 4.1 Benchmark models used in our tests. From left to right: Biplane (6K tris), Small Bunny (16K tris) and Large Bunny (69K tris), Armadillo (345K tris), Dragon (871K tris), Happy Buddha (1.08M tris)

4.1.1 Adaptive Construction Kernel Results

As we have already seen in section 3.2.2 and specifically in Figure 3.3, our first version of the BVH construction has a performance degradation as the input queue grows. To minimize the execution costs in the lower levels of the hierarchy we introduced a adaptive scheme where the work group size was modified in accordance with the average number of primitives per node. To justify the use of that mechanism we demonstrate in Figure 4.2 the differences in terms of performance between both approaches.

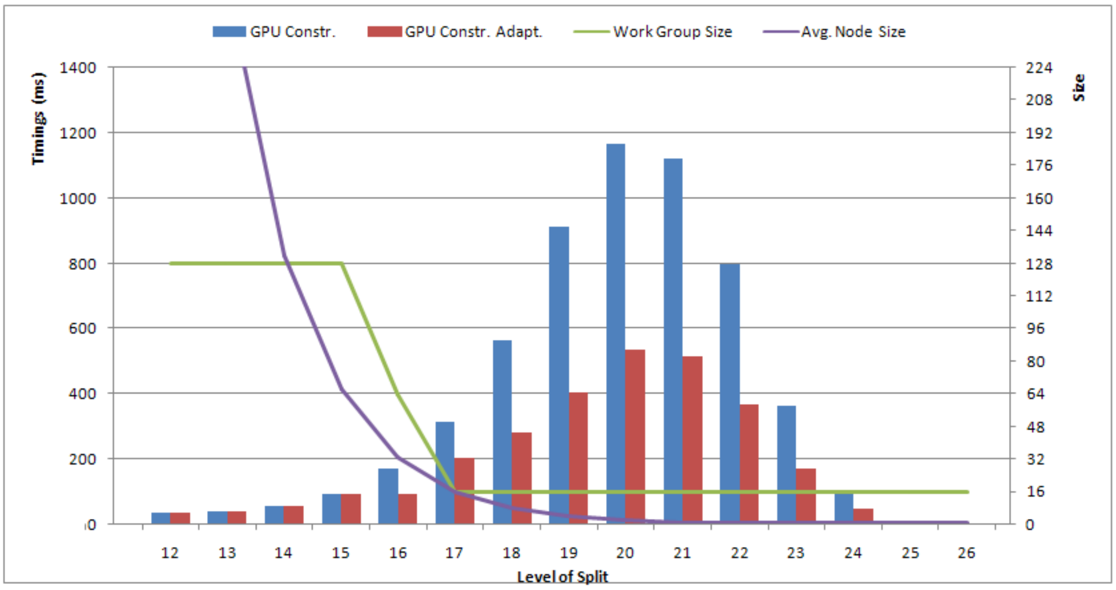


Figure 4.2 Adaptive Kernel timings per level: the adaptive kernel performs better by having less idle threads. The work group size is reduce as the average node size decreases.

The figure shows the timings for the normal and adaptive versions of the construction kernel in each split level and how the group size reduction affects the results achieved. The lines represent the estimated node size and the work group size used for each level. This results were once more obtained by running both kernels to build the Happy Buddha model. We reduced the work group size from an initial value of 128 to a minimum of 16 threads in the last few levels. As expected, the adaptive scheme obtains better results for small nodes, leading to a speedup of 94% from level 16 to 26 (where we reduce the work group size) and an average of 67% for the entire hierarchy. This results refer to the Happy Buddha model but the remaining models follow this tendency and have similar improvements.

Despite this performance upgrade, the adaptive kernel still has some problems. When we reduce the amount of threads per work group to values above 32, we are not using all the threads available, since the cores do not run a full warp (32 threads at a time). Furthermore, even very small nodes are created using parallel algorithms (scan and reduction) in local memory. This memory access overhead is not worth (even for local memory) for few primitives.

4.1.2 Small Splits Construction Kernel Results

To improve even more the construction in the last levels, we applied a change so that each small node is processed by a single thread with no need to synchronization within the group. In Figure 4.3 the results achieved by introducing the small splits optimization are highlighted.

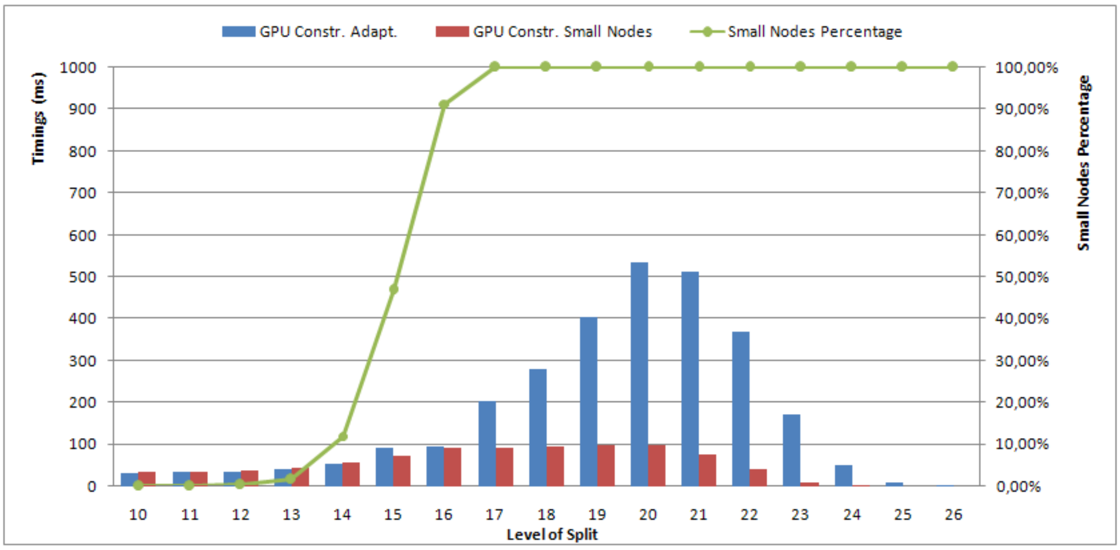


Figure 4.3 Comparison between the Small Splits kernels and the Adaptive Scheme. The first one has better results for small nodes.

After some tests we conclude that for nodes above 55 primitives there is a gain of performance by assigning them to only one thread (with a work group size of 128). When the percentage of small splits exceeds a defined threshold, we switch kernels. From that level to the last level each node is immediately split by a single thread without the filtering process. The percentage threshold chosen was 30% (after several experiments), since it was expected that the remaining 70% of the nodes have an aproximate size.

In the illustrated test (Happy Buddha model) the first 10 levels (starting at level 0) were ignored because the kernels behave in a similar way. What must be examined is the performance gap achieved when we have a substantial amount of small nodes. From level 10 to the end of level 15 that percentage is above 30%, which means that all the small nodes are written in a queue during the normal kernel and processed at the end by single threads. If the percentage is very low we do not have any benefit from it because there is an extra cost of writting and reading nodes from local memory. Only in level 15, where there is a significant number of small nodes (47%) we can obtain a gain in performance (25%) by saving the small nodes in local memory and assigning to each one a single thread.

After the construction of level 15, the remaining levels are processed using a kernel with one thread per node, since we have overcome the threshold barrier. By having all threads working and with few memory accesses we get a faster execution with its peak on level 21 (approximately 7 times faster). The results demonstrate that for our models, in average, the entire hierarchy is constructed from 1,5 to 2 times faster by using the conjugation of the two small splits kernels when compared to the single adaptive kernel and 3 times faster than the normal construction kernel.

4.1.3 Large Nodes Construction Kernel Results

So far we have reduced the construction timings for the lower levels of the hierarchy, where we have a large number of nodes of small size. Now, we will demonstrate the impact achieved by the optimization made to build the first BVH levels. As explained in section 3.2.2.3, our solution was to distribute each large node to several work-groups (as many as multicores). This way we don't have idle threads since all the work groups are running in parallel.

However, this mechanism only performs better that the initial version until a certain level. The number of levels processed with the Large Nodes kernels before switching to the group/node construction varies from model to model and depends of the amount of primitives of each object. In this Large Nodes version, we need to synchronize all the work groups 6 times for each level, corresponding to the 6 kernel invocations (2 kernels for each one of the 3 phases).

In the one group per node kernel, there is no need for synchronization between work groups since each one processes its own node independently. The gain of performance by using the Large Nodes kernels decreases as the ratio triangles per node/nodes decreases, in other words, as the number of primitives per node decreases and the number of synchronizations per level increases. For example, we have used these optimized kernels in the first three levels for the Biplane model, four levels for the 16k Bunny, five levels for the 69k Bunny, six levels for the Armadillo and seven levels for the Dragon and Happy Buddha models. The remaining levels are constructed using the small splits kernels. In Figure 4.4 we expose the difference of timings achieved for the Happy Buddha model. The first seven levels are constructed almost 5 times faster when compared to the simple version, representing an improvement of 70% for the entire hierarchy construction.

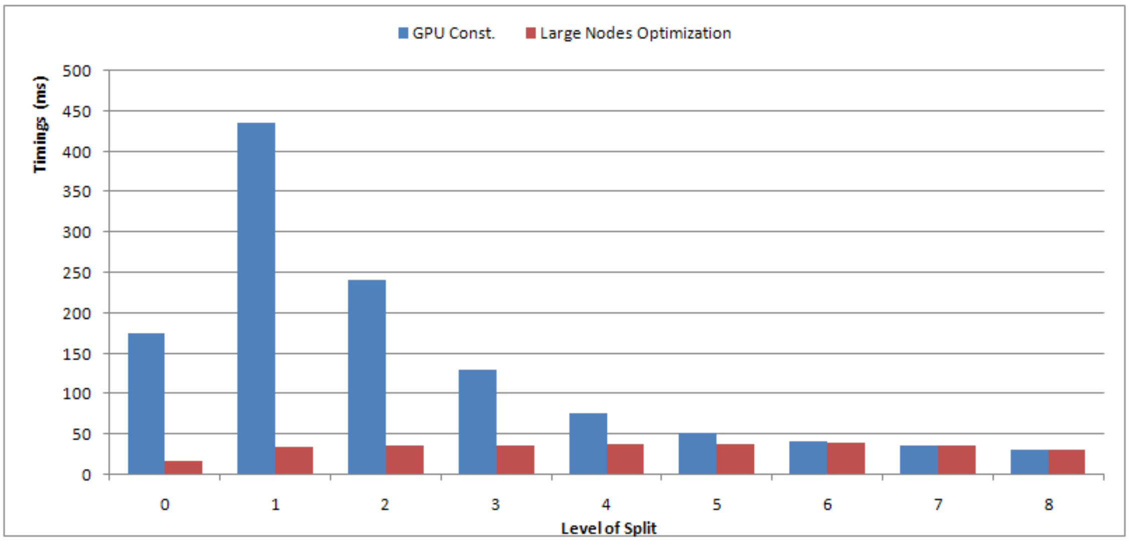


Figure 4.4 Comparison between the Large Nodes kernels and the Small Splits kernels. The first one has better results for large nodes.

We also ran the last version of our construction kernel in another device for comparison. The results listed in Table 4.2 show the difference of performances between the BVH construction phase executed on the CPU and on the GPU, as well as the improvement achieved since the first construction kernel (Normal Construction Kernel) to the last and more optimized version (with large nodes and small splits optimizations).

Table 4.2 Construction phase timings (in ms) for BVH construction of several benchmark models in CPU, GPU with Normal kernel (GPU N. Const.) and with the Large Nodes plus Small Splits optimization (GPU Opt. Const.). In the last column is the performance speedup achieved with the best GPU implementation against the CPU.

Model	Nr. Tris	CPU Const.	GPU N. Const.	GPU Opt. Const.	CPU vs GPU Opt. Speedup
Biplane	6272	85 ms	25 ms	9 ms	9.4×
Small Bunny	16301	221 ms	62 ms	18 ms	12.2×
Large Bunny	69451	1216 ms	269 ms	85 ms	14.3×
Armadillo	345944	6513 ms	1402 ms	502 ms	12.9×
Dragon	871414	14121 ms	3433 ms	1105 ms	12.7×
Happy Buddha	1087716	20875 ms	7189 ms	1376 ms	15.1×

In the first column are the benchmark models (Figure 4.1) used in the tests, followed by their number of primitives and the corresponding construction timings in milliseconds. The last column in the table demonstrates the speed up achieved with the best OpenCL construction kernels when compared with the CPU approach. As we can see the GPU implementation achieves better results than the sequential algorithm. For the tested models, the performance gain ranges between 9 and 15 times and it tends to increase with larger objects Finally, we tested our final construction kernel with the Tesla C2050 and compared the results with the ones obtained with the Quadro FX 3800. Due to the Tesla C2050 characteristics, we were able to increase the work group size to 256 for the construction kernels, in order to achieve a better performance. In Table 4.3 are highlighted the timings with both devices and the performance difference between them. Note that this results refer to the more optimized kernel version.

Table 4.3 Construction phase timings (in ms) for BVH construction of several benchmark models in two different GPUs (Quadro FX 3800 and Tesla C2050). In the last column is the performance speedup achieved by using the second device.

Model	Nr. Tris	Quadro FX 3800	Tesla C2050	Quadro FX 3800 vs Tesla C2050 Speedup
Biplane	6272	9 ms	8 ms	1.1×
Small Bunny	16301	18 ms	10 ms	1.8×
Large Bunny	69451	85 ms	34 ms	2.5×
Armadillo	345944	502 ms	183 ms	2.7×
Dragon	871414	1150 ms	401 ms	2.8×
Happy Buddha	1087716	1376 ms	471 ms	2.9×

4.1.4 Compaction Kernel Results

In Figure 4.5 it is illustrated the time consumed by the compaction phase during the hierarchy construction.

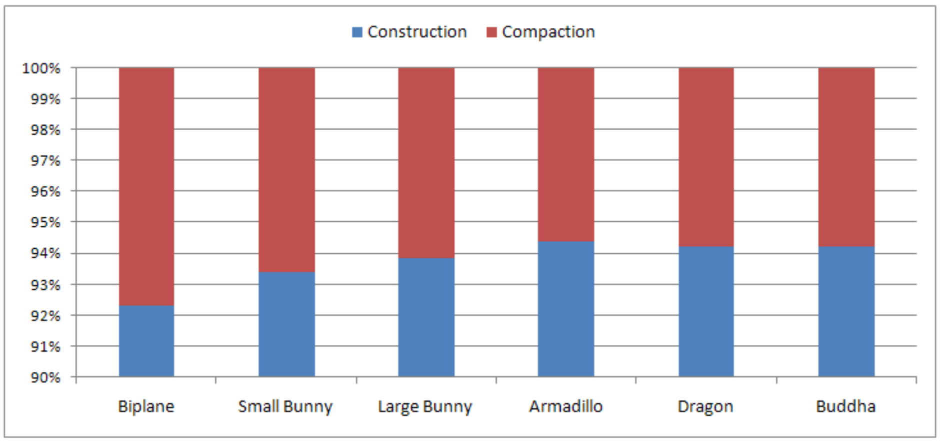


Figure 4.5 Time consumed by the compaction kernel during the entire construction process.

As we can see, the compaction occupies a small portion of the time spent, that tends to decrease with larger models. This was expected since we only invoke the compaction kernel in the levels that have leafs. Furthermore, the kernel uses a highly parallel scan algorithm to compact the array and makes very good use from the GPU resources.

4.2 Broad Phase Collision Detection Kernels Results

4.2.1 Simple Broad Phase Collision Detection Kernel Results

In the Broad Phase we perform a parallel tree traversal in the cloth and object hierarchies. Figure 4.6 demonstrates the performances achieved during this tree traversal between our CPU implementation and the GPU without front kernel.

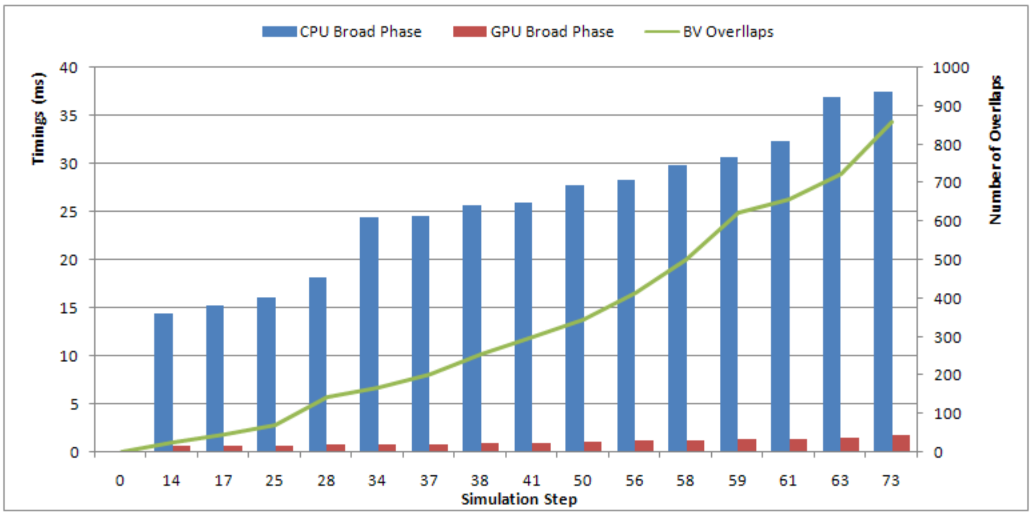


Figure 4.6 Broad Phase kernel performance: CPU and GPU performances during tree traversal during a simulation test. The timings are influenced by the number of collisions found.

The results obtained concern to several discrete time intervals (not necessarily consecutive) during a simulation where we drop a piece of cloth with 2048 triangles on a static object (Large Bunny with 69K triangles), illustrated on Figure 4.7. In the *x* axis are the number of overlaps found in different moments of the simulation and the *y* axis have the execution timings for both algorithms. Obviously, the execution time varies with the number of overlaps found, which also reflects the amount of nodes traversed. The results shown that the OpenCL kernel is much faster than the sequential implementation (about 25 times faster), even running with a single work group.

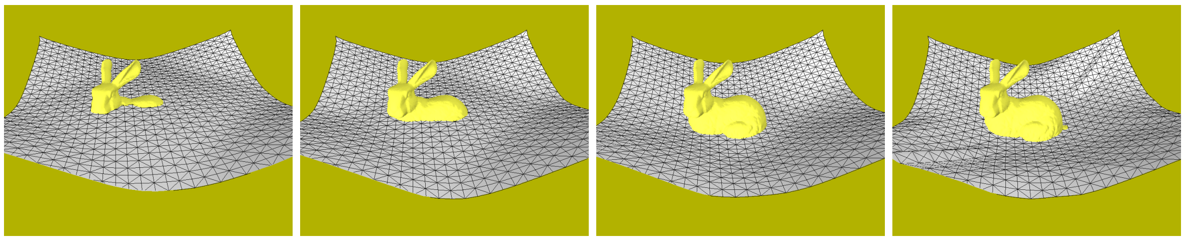


Figure 4.7 Piece of cloth with 2K triangles dropped over a fixed object with 69K triangles. This simulation was used to test both Broad Phase and Narrow Phase collision detection

4.2.2 Front Based Broad Phase Collision Detection Kernel Results

To use several work groups we introduce the front based approach. This algorithm exploits temporal coherence, which means that it works better when the colliding BVs do not vary to much and there are few changes in the front between different time steps. To test the front algorithm performance and compare it with the normal version we run the same simulation illustrated in Figure 4.7 across several time steps (we only show the consecutive steps with more collisions found, to better highlight the differences between the two algorithms) using both implementations. Since our cloth simulation does not have a collision response system for two meshed objects, we have reduced the interval time between successive simulation steps. By doing this we are ensuring that the overlapped BVs in one time step have good chances to collide again in the next step (which result in fewer modifications in the front), as we expect to happen in a real collision. Our results (Figure 4.8) show that the front based implementation has a better overall performance, even with the extra cost of compacting the front nodes and the colliding pairs (ranging between 0.1 and 0.2 ms in this test).

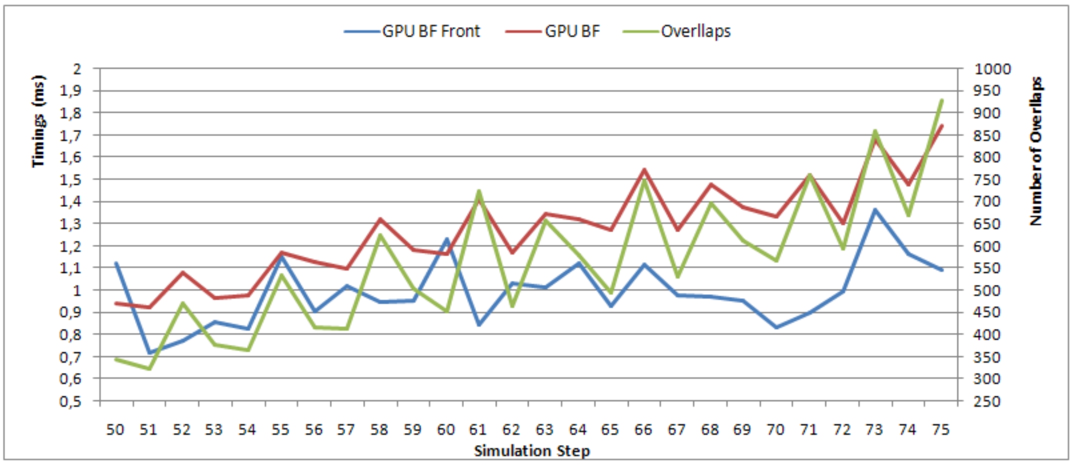


Figure 4.8 Broad Phase with and without the front based approach: comparison between the front based algorithm and the normal version during a collision simulation between a piece of cloth and a static object.

As we can see, the normal broad phase algorithm performance varies with the amount of collisions detected, resulting in more hierarchy levels traversed and pairs of nodes tested. In the other hand, the front based approach does not depend exclusively in the number of collisions, but also in the front variation between successive steps, which explains that in some cases the execution time decreases when the number of overlaps increases.

4.3 Narrow Phase Collision Detection Kernel Results

In the Narrow Phase we perform elementary tests between pairs of primitives in order to find if they are colliding. In our simulator, both CPU and GPU run the same code [22] in these tests. The main difference is that in the CPU we have a single thread to process every pair of primitives, while in the GPU we launch as many threads as the number of elementary tests to be done. As shown in Figure 4.9, this parallelism has a great influence in the final performance. The graph also indicates that as the number of primitive pairs to test increases, the gap between CPU and GPU implementations raises substantially.

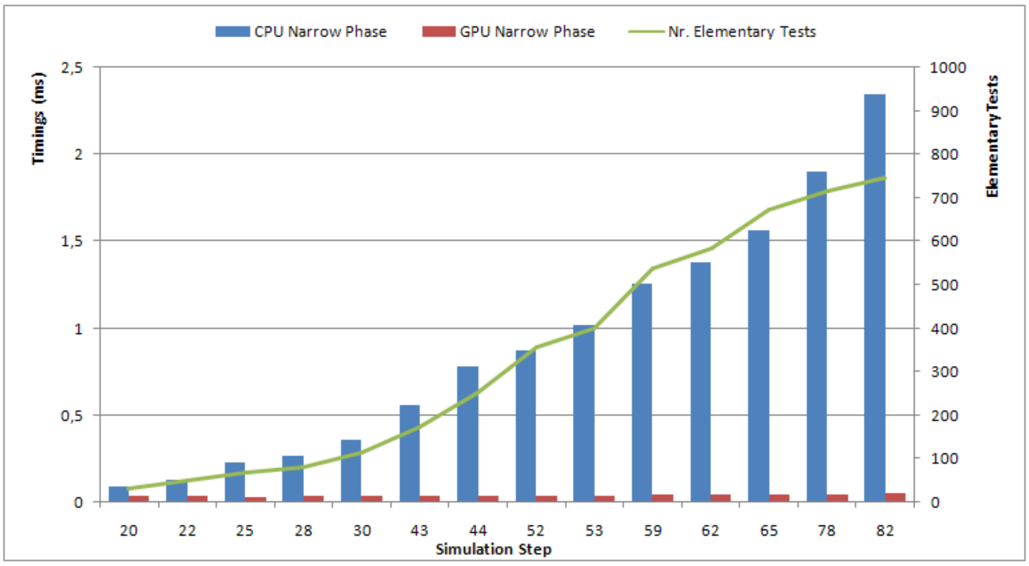


Figure 4.9 Results achieved by both CPU and GPU narrow phase implementations.

4.4 Update Kernel Results

In this section we expose the results achieved by the update kernel. Compared with the analogous CPU implementation, we got an average speedup of 4.5× by performing the update on the GPU. Table 4.4 shows the timings obtained for each of our models and the correspondent speedup. We also concluded that performing the update is 10 tomes faster in average than reconstructing the entire hierarchy. As expected, these results justify the use of the update method, since the BV update is much easier to perform, specially in the higher levels where we only merge pairs of BVs.

Table 4.4 Update phase timings (in ms) for BVH construction of several benchmark models and the performance speedup achieved with the GPU implementation.

Model	Nr. Triangles	CPU Update	GPU Update	Speedup
Biplane	6272	7 ms	2 ms	3.5×
Small Bunny	16301	15 ms	3 ms	5×
Large Bunny	69451	60 ms	12 ms	5×
Armadillo	345944	314 ms	64 ms	4.9×
Dragon	871414	747 ms	159 ms	4.6×
Happy Buddha	1087716	931 ms	201 ms	4.6×

This increase of performance in the higher levels is evidenced on Figure 4.10, where it is shown the execution timings in each level (Happy Buddha model). The time consumed by the kernel is proportional to number of nodes in each level. In addiction, updating the leafs requires to load nodes primitives from global memory, while in the internal nodes we only have to merge their BVs.

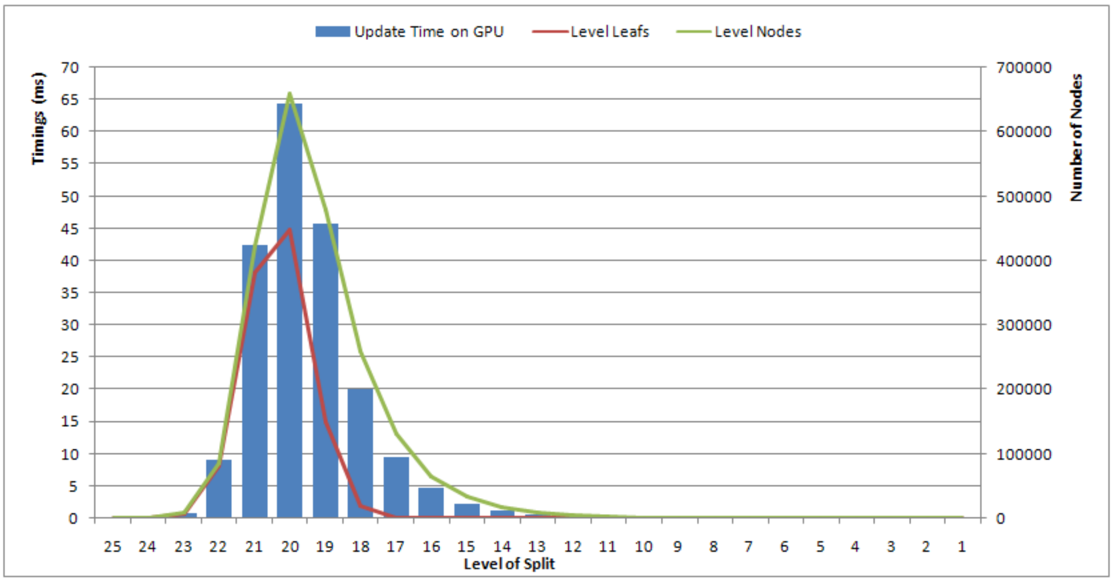


Figure 4.10 Update kernel timings per level: the time consumed is proportional to the number of nodes in each level.

5 . Conclusions and Future Work

Our initial motivation was to accelerate a physics engine through the OpenCL framework. Based on an analysis over some physics engines features, as well as the current GPU implementations in that area, we have chosen the collision detection in cloth simulation as our target.

For that purpose we have implemented a set of kernels to deal with discrete collision detection between a piece of cloth and a meshed object. In one of the kernels we present a way to build Bounding Volume Hierarchies with *k*-Dops on the GPU, using parallel algorithms. We have shown three versions of this construction kernel, explaining the optimizations done and demonstrating the performance gains achieved. We also have proposed two different kernels for the Broad Phase collision detection. The first one corresponds to a hierarchy traversal starting always from the root nodes and performed by a single work group and multiprocessor. The second version implements a front based algorithm that exploits temporal coherence and distributes the work in a more efficient way. To perform the Narrow Phase we adapted an open source code to perform fast triangle-triangle intersection tests on the GPU. Finally, we presented an update kernel using a bottom-up approach that is executed once per level and avoids having to reconstruct the entire hierarchy at every simulation step.

We have tested our construction kernel with several benchmarks and highlighted the difference of performance between the three implemented versions. Furthermore, we have shown the timings achieved for each level of the hierarchy, as well as how much the number of nodes to split influenced the kernel performance and finally what were the kernel bottlenecks. For the same kernel we have demonstrated that the GPU version can perform almost 10 times faster than the analogous CPU approach in our benchmark tests.

Then, a comparison was made between the CPU and GPU Broad Phase algorithms using a simulation with a piece of cloth dropping on a fixed meshed object. Our results revealed that the parallel algorithm is 25 times faster in average than the single core Broad Phase. We have also shown the advantages of using the front based approach and what were its main weaknesses when compared to the normal version.

For the Narrow Phase the difference of execution times between a CPU and a GPU implementation of the same code are evident. Here, the advantage of using the device to perform the elementary intersection tests was clearly visible, specially for large data sets.

Finally, our results revealed that we have a performance gain by executing a BVH update instead of a full rebuild. Once more, we achieved an average gain of 450% by invoking the GPU kernel. We also evidenced the impact of the number of nodes to update in the time consumed by the kernel.

In conclusion, our results have shown that the GPU kernels were much faster than the sequential approach, encouraging the use of this type of device in physics simulations.

Regarding the future work, there are many opportunities to explore. For example, it would be interesting to improve the construction kernel, since it represents the most time consuming phase. In that context, different split strategies could be implemented other than the mean point and test their performance in a parallel environment. For instance, we could split by the median, as explained in section 3.2.1 through the bitonic sort or radix sort algorithms. There is also the possibility to test other methods to choose the plane to split and study their impact on the construction time. In this line of thought, it is not only the construction time that is important. One must also measure eventual benefits from the collision detection phases that may be introduced by different split strategies during construction. Even a slightly heavier construction phase may be better if collision detection times are later drastically decreased.

Regarding the BVs, there are other alternatives, such as AABBs, OBBs or *k*-Dops with a value of *k* larger than 18. By using simpler BVs we could achieve faster construction and update timings but in other hand it would result in more false positives during the Broad Phase. It may be worth since the tree traversal is much less time consuming when compared to the construction.

It would be interesting to use trees with different degrees and measure the impact in the performance. For example, by using quadtrees or octrees we could reduce the number of kernel invocations and distribute more work inside a group during the Broad Phase. With respect to the update kernel, it could be important to implement a mechanism to find if the hierarchy quality is deteriorated enough to be necessary performing a reconstruction.

In addition, we could extend our implementation to work on systems with multiple GPUs taking benefits from the increased availability of computing cores. Finally, it would be interesting to explore the use of the GPU in other physics engines' features, such as continuous collision detection or fluid-cloth collision detection.

A . Appendix

A.1 Reduction and Scan Algorithms

```
/**
 * Reduction algorithm to sum a set of GPU_REAL values.
 * This version adds multiple elements per thread sequentially.
 * This reduces the overall cost of the algorithm while keeping the
 * work complexity O(n) and the step complexity O(log n).
 * (Brent's Theorem optimization)
 */
GPU_REAL reduceSum(__local GPU_REAL* local_data, int blockSize) {

   int tID = get_local_id(0);

   // Do reduction in local memory
   if (blockSize >= 512) {
      if (tID < 256) {
         local_data[tID] += local_data[tID + 256];
      }
      barrier(CLK_LOCAL_MEM_FENCE);
   }
   if (blockSize >= 256) {
      if (tID < 128) {
         local_data[tID] += local_data[tID + 128];
      }
      barrier(CLK_LOCAL_MEM_FENCE);
   }
   if (blockSize >= 128) {
      if (tID <  64) {
         local_data[tID] += local_data[tID +  64];
      }
      barrier(CLK_LOCAL_MEM_FENCE);
   }

   if (tID < 32) {
      if (blockSize >=  64) { local_data[tID] += local_data[tID + 32]; }
      if (blockSize >=  32) { local_data[tID] += local_data[tID + 16]; }
      if (blockSize >=  16) { local_data[tID] += local_data[tID +  8]; }
      if (blockSize >=   8) { local_data[tID] += local_data[tID +  4]; }
      if (blockSize >=   4) { local_data[tID] += local_data[tID +  2]; }
      if (blockSize >=   2) { local_data[tID] += local_data[tID +  1]; }
   }
   barrier(CLK_LOCAL_MEM_FENCE);
   return local_data[0];
}

/**
 * Reduction algorithm to get the minimum/maximum from a set of values
 * For arrowDir == 0 returns the minimum
 * For arrowDir == 1 returns the maximum
 */
GPU_REAL reduceMinMax(__local GPU_REAL* local_data, int arrowDir, int blockSize) {

   int tID = get_local_id(0);

   // Do reduction in local memory
   if (blockSize >= 512) {
      if (tID < 256) {
```

```
      GPU_REAL valA = local_data[tID];
      GPU_REAL valB = local_data[tID + 256];
      if((valB > valA) == arrowDir )
         local_data[tID] = valB;
   }
   barrier(CLK_LOCAL_MEM_FENCE);
}
if (blockSize >= 256) {
   if (tID < 128) {
      GPU_REAL valA = local_data[tID];
      GPU_REAL valB = local_data[tID + 128];
      if((valB > valA) == arrowDir )
         local_data[tID] = valB;
   }
barrier(CLK_LOCAL_MEM_FENCE);
}
if (blockSize >= 128) {
   if (tID <  64) {
      GPU_REAL valA = local_data[tID];
      GPU_REAL valB = local_data[tID + 64];
      if((valB > valA) == arrowDir )
         local_data[tID] = valB;
   }
   barrier(CLK_LOCAL_MEM_FENCE);
}

if (tID < 32) {
   if (blockSize >= 64) {
      GPU_REAL valA = local_data[tID];
      GPU_REAL valB = local_data[tID + 32];
      if((valB > valA) == arrowDir )
         local_data[tID] = valB;
   }
   if (blockSize >= 32) {
      GPU_REAL valA = local_data[tID];
      GPU_REAL valB = local_data[tID + 16];
      if((valB > valA) == arrowDir )
         local_data[tID] = valB;
   }
   if (blockSize >= 16) {
      GPU_REAL valA = local_data[tID];
      GPU_REAL valB = local_data[tID + 8];
      if((valB > valA) == arrowDir )
         local_data[tID] = valB;
   }
   if (blockSize >= 8) {
      GPU_REAL valA = local_data[tID];
      GPU_REAL valB = local_data[tID + 4];
      if((valB > valA) == arrowDir )
         local_data[tID] = valB;
   }
   if (blockSize >= 4) {
      GPU_REAL valA = local_data[tID];
      GPU_REAL valB = local_data[tID + 2];
      if((valB > valA) == arrowDir )
         local_data[tID] = valB;
   }
   if (blockSize >= 2) {
      GPU_REAL valA = local_data[tID];
      GPU_REAL valB = local_data[tID + 1];
      if((valB > valA) == arrowDir )
         local_data[tID] = valB;
      }
   }
   barrier(CLK_LOCAL_MEM_FENCE);
   return local_data[0];
}

/**
 * Prefix Sum algorithm without Shared Memory Bank Conflicts
 */
void scan(__local int* scanArray, int n) {

   int threadID = get_local_id(0);
   int offset = 1;

   // Build sum in place up the tree
   for (int d = n>>1; d > 0; d >>= 1) {
   barrier(CLK_LOCAL_MEM_FENCE);
      if (threadID < d) {
         int ai = offset*(2*threadID+1)-1;
         int bi = offset*(2*threadID+2)-1;
         ai += CONFLICT_FREE_OFFSET(ai);
         bi += CONFLICT_FREE_OFFSET(bi);
         scanArray[bi] += scanArray[ai];
      }
      offset *= 2;
   }

   if (threadID == 0)
      scanArray[n - 1 + CONFLICT_FREE_OFFSET(n - 1)] = 0;

   // Traverse down tree and build scan
   for (int d = 1; d < n; d *= 2) {
      offset >>= 1;
      barrier(CLK_LOCAL_MEM_FENCE);
      if (threadID < d) {
         int ai = offset*(2*threadID+1)-1;
         int bi = offset*(2*threadID+2)-1;
         ai += CONFLICT_FREE_OFFSET(ai);
         bi += CONFLICT_FREE_OFFSET(bi);
         int t = scanArray[ai];
         scanArray[ai] = scanArray[bi];
         scanArray[bi] += t;
      }
   }
}
```

A.2 Common Functions

```
// Structure to define a triangle
typedef struct {
   GPU_REAL v0x;
   GPU_REAL v0y;
   GPU_REAL v0z;

   GPU_REAL v1x;
   GPU_REAL v1y;
   GPU_REAL v1z;

   GPU_REAL v2x;
   GPU_REAL v2y;
   GPU_REAL v2z;
```

```
} gpu_triangle;

// Structure to define a BVH node
typedef struct {
   GPU_REAL kDop[KDOP_SIZE];
   int id;
   int parent;
   int leftChild;
   int rightChild;
   int beginTri;
   int endTri;
   int level;
} kNode;

unsigned int nextPow2(unsigned int x) {
   --x;
   x |= x >> 1;
   x |= x >> 2;
   x |= x >> 4;
   x |= x >> 8;
   x |= x >> 16;
   return ++x;
}

bool isLeaf(kNode node) {
   return (node.leftChild == -1 && node.rightChild == -1);
}

bool overlap(GPU_REAL* kDopA, GPU_REAL* kDopB) {
   for(int i = 0, j = HALF_KDOP_SIZE; i < HALF_KDOP_SIZE && j < KDOP_SIZE; i++, j++) {
      if(kDopA[i] > kDopB[j])
         return false;
      if(kDopB[i] > kDopA[j])
         return false;
   }
   return true;
}

GPU_REAL triangleCentroid(int direction, gpu_triangle triangle) {
   GPU_REAL v0, v1, v2;
   GPU_REAL centroid;
   switch (direction) {
   case X:
      v0 = triangle.v0x;
      v1 = triangle.v1x;
      v2 = triangle.v2x;
      centroid = (v0 + v1 + v2) / 3.0;
      return centroid;
      break;
   case Y:
      v0 = triangle.v0y;
      v1 = triangle.v1y;
      v2 = triangle.v2y;
      centroid = (v0 + v1 + v2) / 3.0;
      return centroid;
      break;
   case Z:
      v0 = triangle.v0z;
      v1 = triangle.v1z;
      v2 = triangle.v2z;
      centroid = (v0 + v1 + v2) / 3.0;
      return centroid;
```

```
         break;
      case XY:
         v0 = triangle.v0x + triangle.v0y;
         v1 = triangle.v1x + triangle.v1y;
         v2 = triangle.v2x + triangle.v2y;
         centroid = (v0 + v1 + v2) / 3.0;
         return centroid;
         break;
      case XZ:
         v0 = triangle.v0x + triangle.v0z;
         v1 = triangle.v1x + triangle.v1z;
         v2 = triangle.v2x + triangle.v2z;
         centroid = (v0 + v1 + v2) / 3.0;
         return centroid;
         break;
      case YZ:
         v0 = triangle.v0y + triangle.v0z;
         v1 = triangle.v1y + triangle.v1z;
         v2 = triangle.v2y + triangle.v2z;
         centroid = (v0 + v1 + v2) / 3.0;
         return centroid;
         break;
      case XMY:
         v0 = triangle.v0x - triangle.v0y;
         v1 = triangle.v1x - triangle.v1y;
         v2 = triangle.v2x - triangle.v2y;
         centroid = (v0 + v1 + v2) / 3.0;
         return centroid;
         break;
      case XMZ:
         v0 = triangle.v0x - triangle.v0z;
         v1 = triangle.v1x - triangle.v1z;
         v2 = triangle.v2x - triangle.v2z;
         centroid = (v0 + v1 + v2) / 3.0;
         return centroid;
         break;
      case YMZ:
         v0 = triangle.v0y - triangle.v0z;
         v1 = triangle.v1y - triangle.v1z;
         v2 = triangle.v2y - triangle.v2z;
         centroid = (v0 + v1 + v2) / 3.0;
         return centroid;
         break;
      }
   return 0;
}

GPU_REAL getOverallCentroid(int beginTri, int endTri, __global gpu_triangle* triangles,
__global int* trianglesIndexes, int splitDirection, __local GPU_REAL* reductionArray) {

   GPU_REAL acum = 0, centroid;
   int localID = get_local_id(0);
   int localSize = get_local_size(0);
   int nTriangles = endTri - beginTri + 1;
   int blockSize = localSize;
   int readIndex = beginTri + localID;

   while(readIndex < endTri) {
      int triangleIndex = trianglesIndexes[readIndex];
      gpu_triangle triangle = triangles[triangleIndex];
      centroid = triangleCentroid(splitDirection, triangle);
      acum += centroid;
      readIndex += localSize;
   }

   // Try to minimize algorithm steps
   if(nTriangles < localSize)
      blockSize = nextPow2(nTriangles);
   if(localID < blockSize)
      reductionArray[localID] = acum;

   barrier(CLK_LOCAL_MEM_FENCE);

   // Reduction Algorithm
   GPU_REAL sum = reduceSum(reductionArray, blockSize);
   return sum / (GPU_REAL)nTriangles;
}

GPU_REAL getOverallCentroidSingleThread(int beginTri, int endTri,
__global gpu_triangle* triangles, __global int* trianglesIndexes,
int splitDirection) {

   GPU_REAL acum = 0, centroid;
   int nTriangles = endTri - beginTri + 1;
   for(int i = beginTri; i <= endTri; i++) {
      int triangleIndex = trianglesIndexes[i];
      gpu_triangle triangle = triangles[triangleIndex];
      centroid = triangleCentroid(splitDirection, triangle);
      acum += centroid;
   }
   return acum / (GPU_REAL)nTriangles;
}

/**
 * Return the longest k-Dop direction
 * from k/2 possibilities
 */
int getLongestDirection(GPU_REAL* k) {

   GPU_REAL longestDirectionValue = k[9] - k[0];
   int longestDirection = 0;
   for(int i = 1; i < HALF_KDOP_SIZE; i++) {
      GPU_REAL lDValue = k[i+HALF_KDOP_SIZE] - k[i];
      if(lDValue > longestDirectionValue) {
         longestDirectionValue = lDValue;
         longestDirection = i;
      }
   }
   return longestDirection;
}

void build18DopFromBlockTriangles(GPU_REAL* k_current, gpu_triangle triangle,
__local GPU_REAL* local_data) {

   unsigned int local_size = get_local_size(0);
   unsigned int lid = get_local_id(0);

   GPU_REAL x0 = triangle.v0x; GPU_REAL x1 = triangle.v1x; GPU_REAL x2 = triangle.v2x;
   GPU_REAL y0 = triangle.v0y; GPU_REAL y1 = triangle.v1y; GPU_REAL y2 = triangle.v2y;
   GPU_REAL z0 = triangle.v0z; GPU_REAL z1 = triangle.v1z; GPU_REAL z2 = triangle.v2z;

   // x
   GPU_REAL v_min = fmin(x0, x1); v_min = fmin(v_min, x2);
   local_data[lid] = v_min;
```

```
k_current[0] = reduceMinMax(local_data, DESCENDING, local_size);
GPU_REAL v_max = fmax(x0, x1); v_max = fmax(v_max, x2);
local_data[lid] = v_max;
k_current[9] = reduceMinMax(local_data, ASCENDING, local_size);

// y
v_min = fmin(y0, y1); v_min = fmin(v_min, y2);
local_data[lid] = v_min;
k_current[1] = reduceMinMax(local_data, DESCENDING, local_size);
v_max = fmax(y0, y1); v_max = fmax(v_max, y2);
local_data[lid] = v_max;
k_current[10] = reduceMinMax(local_data, ASCENDING, local_size);

// z
v_min = fmin(z0, z1); v_min = fmin(v_min, z2);
local_data[lid] = v_min;
k_current[2] = reduceMinMax(local_data, DESCENDING, local_size);
v_max = fmax(z0, z1); v_max = fmax(v_max, z2);
local_data[lid] = v_max;
k_current[11] = reduceMinMax(local_data, ASCENDING, local_size);

// x + y
v_min = fmin(x0+y0, x1+y1); v_min = fmin(v_min, x2+y2);
local_data[lid] = v_min;
k_current[3] = reduceMinMax(local_data, DESCENDING, local_size);
v_max = fmax(x0+y0, x1+y1); v_max = fmax(v_max, x2+y2);
local_data[lid] = v_max;
k_current[12] = reduceMinMax(local_data, ASCENDING, local_size);

// x + z
v_min = fmin(x0+z0, x1+z1); v_min = fmin(v_min, x2+z2);
local_data[lid] = v_min;
k_current[4] = reduceMinMax(local_data, DESCENDING, local_size);
v_max = fmax(x0+z0, x1+z1); v_max = fmax(v_max, x2+z2);
local_data[lid] = v_max;
k_current[13] = reduceMinMax(local_data, ASCENDING, local_size);

// y + z
v_min = fmin(y0+z0, y1+z1); v_min = fmin(v_min, y2+z2);
local_data[lid] = v_min;
k_current[5] = reduceMinMax(local_data, DESCENDING, local_size);
v_max = fmax(y0+z0, y1+z1); v_max = fmax(v_max, y2+z2);
local_data[lid] = v_max;
k_current[14] = reduceMinMax(local_data, ASCENDING, local_size);

// x - y
v_min = fmin(x0-y0, x1-y1); v_min = fmin(v_min, x2-y2);
local_data[lid] = v_min;
k_current[6] = reduceMinMax(local_data, DESCENDING, local_size);
v_max = fmax(x0-y0, x1-y1); v_max = fmax(v_max, x2-y2);
local_data[lid] = v_max;
k_current[15] = reduceMinMax(local_data, ASCENDING, local_size);

// x - z
v_min = fmin(x0-z0, x1-z1); v_min = fmin(v_min, x2-z2);
local_data[lid] = v_min;
k_current[7] = reduceMinMax(local_data, DESCENDING, local_size);
v_max = fmax(x0-z0, x1-z1); v_max = fmax(v_max, x2-z2);
local_data[lid] = v_max;
k_current[16] = reduceMinMax(local_data, ASCENDING, local_size);

// y - z
```

```
   v_min = fmin(y0-z0, y1-z1); v_min = fmin(v_min, y2-z2);
   local_data[lid] = v_min;
   k_current[8] = reduceMinMax(local_data, DESCENDING, local_size);
   v_max = fmax(y0-z0, y1-z1); v_max = fmax(v_max, y2-z2);
   local_data[lid] = v_max;
   k_current[17] = reduceMinMax(local_data, ASCENDING, local_size);
}

void build18DopFromTriangle(GPU_REAL* k, gpu_triangle triangle) {

   GPU_REAL x0 = triangle.v0x; GPU_REAL x1 = triangle.v1x; GPU_REAL x2 = triangle.v2x;
   GPU_REAL y0 = triangle.v0y; GPU_REAL y1 = triangle.v1y; GPU_REAL y2 = triangle.v2y;
   GPU_REAL z0 = triangle.v0z; GPU_REAL z1 = triangle.v1z; GPU_REAL z2 = triangle.v2z;

   // x
   k[0] = fmin(x0, x1); k[0] = fmin(k[0], x2); k[0] -= EPSILON;
   k[9] = fmax(x0, x1); k[9] = fmax(k[9], x2);  k[9] += EPSILON;

   // y
   k[1] = fmin(y0, y1); k[1] = fmin(k[1], y2); k[1] -= EPSILON;
   k[10] = fmax(y0, y1); k[10] = fmax(k[10], y2); k[10] += EPSILON;

   // z
   k[2] = fmin(z0, z1); k[2] = fmin(k[2], z2); k[2] -= EPSILON;
   k[11] = fmax(z0, z1); k[11] = fmax(k[11], z2); k[11] += EPSILON;

   // x + y
   k[3] = fmin(x0+y0, x1+y1); k[3] = fmin(k[3], x2+y2); k[3] -= EPSILON;
   k[12] = fmax(x0+y0, x1+y1); k[12] = fmax(k[12], x2+y2); k[12] += EPSILON;

   // x + z
   k[4] = fmin(x0+z0, x1+z1); k[4] = fmin(k[4], x2+z2); k[4] -= EPSILON;
   k[13] = fmax(x0+z0, x1+z1); k[13] = fmax(k[13], x2+z2); k[13] += EPSILON;

   // y + z
   k[5] = fmin(y0+z0, y1+z1); k[5] = fmin(k[5], y2+z2); k[5] -= EPSILON;
   k[14] = fmax(y0+z0, y1+z1); k[14] = fmax(k[14], y2+z2); k[14] += EPSILON;

   // x - y
   k[6] = fmin(x0-y0, x1-y1); k[6] = fmin(k[6], x2-y2); k[6] -= EPSILON;
   k[15] = fmax(x0-y0, x1-y1); k[15] = fmax(k[15], x2-y2); k[15] += EPSILON;

   // x - z
   k[7] = fmin(x0-z0, x1-z1); k[7] = fmin(k[7], x2-z2); k[7] -= EPSILON;
   k[16] = fmax(x0-z0, x1-z1); k[16] = fmax(k[16], x2-z2); k[16] += EPSILON;

   // y - z
   k[8] = fmin(y0-z0, y1-z1); k[8] = fmin(k[8], y2-z2); k[8] -= EPSILON;
   k[17] = fmax(y0-z0, y1-z1); k[17] = fmax(k[17], y2-z2); k[17] += EPSILON;
}

void build18DopSingleThread(kNode* node, __global gpu_triangle* triangles,
__global int* tri_indices, int beginTri, int endTri) {

   GPU_REAL k[KDOP_SIZE];
   for(int t = beginTri; t <= endTri; t++) {
      int tri_idx = tri_indices[t];
      gpu_triangle triangle = triangles[tri_idx];

      build18DopFromTriangle(&k, triangle);

      if(t == beginTri) {
            for(int i=0, j = HALF_KDOP_SIZE; i < HALF_KDOP_SIZE && j < KDOP_SIZE; i++, j++)
            {
                (*node).kDop[i] = k[i];
                (*node).kDop[j] = k[j];
            }
        }
        else {
            for(int i=0, j=HALF_KDOP_SIZE; i < HALF_KDOP_SIZE && j < KDOP_SIZE; i++, j++)
            {
                (*node).kDop[i] = fmin(k[i], (*node).kDop[i]);
                (*node).kDop[j] = fmax(k[j], (*node).kDop[j]);
            }
        }
    }
}

void build18Dop(kNode *node, int beginTri, int endTri, __global gpu_triangle* triangles,
__global int* trianglesIndexesOuput__local GPU_REAL* reductionArray) {

    GPU_REAL k[KDOP_SIZE];
    int triangleIndex;
    unsigned int localID = get_local_id(0);
    unsigned int localSize = get_local_size(0);
    int numTriangles = endTri - beginTri + 1;
    int trianglesPerThread = numTriangles / localSize;
    if(numTriangles % localSize != 0)
        trianglesPerThread++;

    for(int t = 0; t < trianglesPerThread; t++) {
        int index = beginTri + localID + t * localSize;
        if(index > endTri)
            triangleIndex = trianglesIndexesOuput[beginTri];
        else
            triangleIndex = trianglesIndexesOuput[index];
        gpu_triangle triangle = triangles[triangleIndex];

        // Compute a 18-Dop for each block of triangles
        build18DopFromBlockTriangles(&k, triangle, reductionArray);

        // Thread 0 updates the k-Dop with the current values
        if(localID == 0) {
            if(t == 0) {
                for(int i=0, j=HALF_KDOP_SIZE; i < HALF_KDOP_SIZE && j < KDOP_SIZE; i++, j++)
                {
                    (*node).kDop[i] = k[i];
                    (*node).kDop[j] = k[j];
                }
            }
            else {
                for(int i=0, j=HALF_KDOP_SIZE; i < HALF_KDOP_SIZE && j < KDOP_SIZE; i++, j++)
                {
                    (*node).kDop[i] = fmin(k[i], (*node).kDop[i]);
                    (*node).kDop[j] = fmax(k[j], (*node).kDop[j]);
                }
            }
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}

int splitByCentroid(int beginTri, int endTri, __global gpu_triangle* triangles,
__global int* trianglesIndexesInput, __global int* trianglesIndexesOutput,
GPU_REAL overallCentroid, int splitDirection, __local int* scanArray,
__local int* lastProcessedFlag) {

    int totalPositives = 0, negatives = 0, blockPositives, lastProcessedIndex, to_process;
    int readIndex, triangleIndex, flag, newTriPos;
    GPU_REAL centroid;
    int localID = get_local_id(0);
    int localSize = get_local_size(0);

    int numTriangles = endTri - beginTri + 1;
    int triangles_per_thread = numTriangles / localSize;
    int n = localSize;
    int e = numTriangles % localSize;
    if(e != 0) {
        triangles_per_thread++;
        n = nextPow2(e);
    }

    for(int i = 0; i < triangles_per_thread; i++) {
        readIndex = beginTri + localID + i * localSize;
        if(readIndex <= endTri) {
            triangleIndex = trianglesIndexesInput[readIndex];
            gpu_triangle triangle = triangles[triangleIndex];
            GPU_REAL centroid = triangleCentroid(splitDirection, triangle);
            if(centroid < overallCentroid)
                flag = 1;
            else
                flag = 0;
        }
        else
            flag = 0;

        int bankOffset = CONFLICT_FREE_OFFSET(localID);
        scanArray[localID + bankOffset] = flag;

        to_process = numTriangles - localSize * i;
        lastProcessedIndex = min(localSize-1, to_process-1);
        if(localID == lastProcessedIndex)
            (*lastProcessedFlag) = flag;

        // Scan Algorithm
        if(i == triangles_per_thread-1)
            scan(scanArray, n);
        else
            scan(scanArray, localSize);
        barrier(CLK_LOCAL_MEM_FENCE);

        // Sorting Process
        blockPositives = scanArray[lastProcessedIndex] + (*lastProcessedFlag);
        negatives += lastProcessedIndex + 1 - blockPositives;
        if(readIndex <= endTri) {
            // Triangles that will be placed in the left child
            if(flag == 1)
                newTriPos = beginTri + scanArray[localID + bankOffset] + totalPositives;
            // Triangles that will be placed in the right child
            else
                newTriPos = localID-scanArray[localID+bankOffset] + (endTri+1-negatives);
            // Saving indexes' new order
            trianglesIndexesOutput[newTriPos] = triangleIndex;
        }
        totalPositives += blockPositives;
    }
    return beginTri + totalPositives;
}

int splitByCentroidSingleThread(int beginTri, int endTri,
__global gpu_triangle* triangles, __global int* trianglesIndexesInput,
__global int* trianglesIndexesOutput, GPU_REAL overallCentroid, int splitDirection) {

    int positives = 0, negatives = 0, triangleIndex;
    GPU_REAL centroid;

    for(int i = beginTri; i <= endTri; i++) {
        triangleIndex = trianglesIndexesInput[i];
        gpu_triangle triangle = triangles[triangleIndex];
        centroid = triangleCentroid(splitDirection, triangle);

        if(centroid < overallCentroid) {
            trianglesIndexesOutput[beginTri + positives] = triangleIndex;
            positives++;
        }
        else {
            trianglesIndexesOutput[endTri - negatives] = triangleIndex;
            negatives++;
        }
    }
    return beginTri + positives;
}

void setNodeInfo(kNode *node, int beginTri, int endTri, int parentID, int nodeID,
int level, int numLeafs, int trianglesPerNode) {

    int numTriangles = endTri - beginTri + 1;
    (*node).id = nodeID;
    (*node).parent = parentID;
    if(numTriangles <= trianglesPerNode) {
        (*node).leftChild = -1;
        (*node).rightChild = -1;
    }
    else {
        (*node).leftChild = (nodeID - numLeafs) * 2 + 1;
        (*node).rightChild = (nodeID - numLeafs) * 2 + 2;
    }
    (*node).beginTri = beginTri;
    (*node).endTri = endTri;
    (*node).level = level + 1;
}

/*
 * Kernel to initialize the array with triangles' indexes
 */
__kernel void initTriangleIndexes(__global int* triIndexes, int nTriangles) {
    int threadID = get_global_id(0);
    if(threadID < nTriangles)
        triIndexes[threadID] = threadID;
}
```

A.3 BVH Construction Kernels

A.3.1 BVH Root Construction Kernel

```
/**
```

```
 * Kernel to build the BVH root executed by a single Work-Group
 * Arguments:
 * triangle* triangles - object's triangles
 * trianglesIndexes - primitives' indexes
 * hierarchy - Array with the current hierarchy
 * outputQueue - Queue where is written the root node
 * trianglesPerNode - Minimum number of triangles per node
 * nTriangles - Total number of object's triangles
 */
__kernel void buildBVHRoot(__global gpu_triangle* triangles,
__global int* trianglesIndexes, __global kNode* hierarchy,
__global kNode* outputQueue, int trianglesPerNode, int nTriangles) {

   // k-Dop Construction Process executed by a single Work-Group
   if(get_group_id(0) == 0) {
      // GPU_REAL local array to perform parallel reductions
      __local GPU_REAL localGpuRealArray[WORKGROUPSIZE*2];
      kNode root;
      // Build the root's k-Dop
      build18Dop(&root,0,nTriangles-1,trianglesIndexes,triangles,localGpuRealArray);
      if(get_local_id(0) == 0) {
         setNodeInfo(&root, 0, nTriangles-1, -1, 0, -1, 0, trianglesPerNode);
         hierarchy[0] = root;
         outputQueue[0] = root;
      }
   }
}
```

A.3.2 BVH Construction Kernel

```
__kernel void buildBVHNormal(__global gpu_triangle* triangles,
__global kNode* hierarchy, __global int* trianglesIndexesInput,
__global int* trianglesIndexesOutput, __global kNode* inputQueue,
__global kNode* outputQueue, int queueSize, int nTriangles,
int trianglesPerNode, __global int* leafs, __global int* smallNodesCounter) {

   unsigned int localSize = get_local_size(0);
   unsigned int groupID = get_group_id(0);
   unsigned int localID = get_local_id(0);

   __local int localIntArray[WORKGROUPSIZE];
   __local GPU_REAL localGpuRealArray[WORKGROUPSIZE];

   __local kNode splitNodeLocal;
   __local int lastProcessedFlag;
   __local int leafCounter;

   kNode splitNode, left, right;
   int privateLeafFlag = -1;

   if(localID == 0)
      leafCounter = 0;

   int dequeueIndex = groupID;
   while(dequeueIndex < queueSize) {
      if(localID == 0) // read a split object from the input queue
         splitNodeLocal = inputQueue[dequeueIndex];
      barrier(CLK_LOCAL_MEM_FENCE);
      splitNode = splitNodeLocal;

      int beginTri = splitNode.beginTri;
      int endTri = splitNode.endTri;
      int numTriangles = endTri - beginTri + 1;

      // Choose the split axis by the longest direction method
      int splitDirection = getLongestDirection(splitNode.kDop);
      // Compute k-Dop split value by the mean method
      GPU_REAL overallCentroid = getOverallCentroid(beginTri, endTri, triangles,
      trianglesIndexesInput, splitDirection, localGpuRealArray);
      // Split and reorder triangles' indices
      int splitPoint = splitByCentroid(beginTri, endTri, triangles,
      trianglesIndexesInput, trianglesIndexesOutput, overallCentroid,
      splitDirection, localIntArray, &lastProcessedFlag);
      // Commit triangles' indexes to global memory to guarantee that all the threads
      // in this Work-Group read updated data during the k-Dop construction
      barrier(CLK_GLOBAL_MEM_FENCE);

      int leftTriangles = splitPoint - beginTri;
      int rightTriangles = (endTri+1) - splitPoint;

      // If the subdivision led to the same geometry as the current node,
      // we simply give half primitives for each node
      if((leftTriangles == 0) || (rightTriangles == 0))
         splitPoint = beginTri + (numTriangles/2);

      // k-Dop Construction Process
      build18Dop(&left, beginTri, splitPoint-1, trianglesIndexesOutput, triangles,
      localGpuRealArray);
      barrier(CLK_LOCAL_MEM_FENCE);

      build18Dop(&right, splitPoint, endTri, trianglesIndexesOutput, triangles,
      localGpuRealArray);
      barrier(CLK_LOCAL_MEM_FENCE);

      if(localID == 0) {
         setNodeInfo(&left, beginTri, splitPoint-1, splitNode.id, splitNode.leftChild,
         splitNode.level, leafs[1], trianglesPerNode);
         setNodeInfo(&right, splitPoint, endTri, splitNode.id, splitNode.rightChild,
         splitNode.level, leafs[1], trianglesPerNode);

      if(isLeaf(left) || isLeaf(right))
         if(privateLeafFlag < 0)
            privateLeafFlag = atom_inc(&leafCounter);

         hierarchy[splitNode.leftChild] = left;
         hierarchy[splitNode.rightChild] = right;
         outputQueue[dequeueIndex*2] = left;
         outputQueue[dequeueIndex*2+1] = right;
      }
      dequeueIndex += get_num_groups(0);
   }
   if(privateLeafFlag == 0)
      leafs[0] = 1; // flag to indicate that there are leaf nodes in this level
}
```

A.3.3 Small Splits Construction Kernels

```
/**
 * Kernel to build a BVH in which each node is split by a single thread
 * Arguments:
 * triangle* triangles - object's triangles
 * trianglesIndexesInput - primitives' indexes before the split reordering
 * trianglesIndexesOutput - primitives' indexes after the split reordering
 * hierarchy - Array with the current hierarchy
 * inputQueue - Queue where the split nodes are read
 * outputQueue - Queue where are written the new generated nodes
 * queueSize - Size of the Input Queue
 * trianglesPerNode - Minimum number of triangles per node
 * nTriangles - Total number of object’s triangles
 * leafs - Flag used to signal the existence of leafs in the current level
 * smallNodesCounter - Counts the number of small nodes (not used in this kernel)
 */
__kernel void buildBVHSingleThread(__global gpu_triangle* triangles,
__global int* trianglesIndexesInput, __global int* trianglesIndexesOutput,
__global kNode* hierarchy, __global kNode* inputQueue, __global kNode* outputQueue,
int queueSize, int trianglesPerNode, int nTriangles, __global int* leafs,
__global int* smallNodesCounter) {

   kNode splitObject, left, right;
   int privateLeafFlag = -1;
   __local int leafCounter;

   if(get_local_id(0) == 0)
      leafCounter = 0;

   int dequeueIndex = get_global_id(0);
   while(dequeueIndex < queueSize) {
      // Read the node to split from the Input Queue
      splitObject = inputQueue[dequeueIndex];
      int beginTri = splitObject.beginTri;
      int endTri = splitObject.endTri;
      int numTriangles = endTri - beginTri + 1;

      // Choose the split axis by the longest direction method
      int       splitDirection = getLongestDirection(splitObject.kDop);
      // Compute k-Dop split value by the mean method
      GPU_REAL overallCentroid = getOverallCentroidSingleThread(beginTri, endTri,
      triangles, trianglesIndexesInput, splitDirection);
      // Split and reorder triangles’ indices
      int splitPoint = splitByCentroidSingleThread(beginTri, endTri, triangles,
      trianglesIndexesInput, trianglesIndexesOutput, overallCentroid, splitDirection);

      int leftTriangles = splitPoint - beginTri;
      int rightTriangles = (endTri+1) - splitPoint;

      // If the subdivision led to the same geometry as the current node,
      // we simply give half primitives for each node
      if((leftTriangles == 0) || (rightTriangles == 0))
         splitPoint = beginTri + (numTriangles/2);

      // k-Dop Construction Process
      build18DopSingleThread(&left, triangles, trianglesIndexesOutput,
      beginTri, splitPoint-1);
      build18DopSingleThread(&right, triangles, trianglesIndexesOutput,
      splitPoint, endTri);

      setNodeInfo(&left, beginTri, splitPoint-1, splitObject.id,
      splitObject.leftChild, splitObject.level, leafs[1], trianglesPerNode);
      setNodeInfo(&right, splitPoint, endTri, splitObject.id,
      splitObject.rightChild, splitObject.level, leafs[1], trianglesPerNode);

      if(isLeaf(left) || isLeaf(right))
         if(privateLeafFlag < 0)
            privateLeafFlag = atom_inc(&leafCounter);

      hierarchy[splitObject.leftChild] = left;
      hierarchy[splitObject.rightChild] = right;
      outputQueue[dequeueIndex*2] = left;
      outputQueue[dequeueIndex*2+1] = right;
      dequeueIndex += get_global_size(0);
   }
   // The first thread to find a leaf sets the leaf flag to true
   if(privateLeafFlag == 0) {
      // Flag to indicate that there are leaf nodes in this
      // level and we need to perform a Compaction
         leafs[0] = 1;
   }
}

/**
 * Kernel to build a BVH in which each node is split by a
 * Work-Group and the small nodes are written into a local
 * queue to be processed individually by single threads
 * Arguments:
 * triangle* triangles - object's triangles
 * trianglesIndexesInput - primitives' indexes before the split reordering
 * trianglesIndexesOutput - primitives' indexes after the split reordering
 * hierarchy - Array with the current hierarchy
 * inputQueue - Queue where the split nodes are read
 * outputQueue - Queue where are written the new generated nodes
 * queueSize - Size of the Input Queue
 * trianglesPerNode - Minimum number of triangles per node
 * nTriangles - Total number of object's triangles
 * leafs - Flag used to signal the existence of leafs in the current level
 * smallNodesCounter - Counts the number of small nodes
 */
__kernel void buildBVH(__global gpu_triangle* triangles,
__global int* trianglesIndexesInput, __global int* trianglesIndexesOutput,
__global kNode* hierarchy, __global kNode* inputQueue,
__global kNode* outputQueue, int queueSize, int trianglesPerNode, int nTriangles,
__global int* leafs, __global int* smallNodesCounter) {

   unsigned int localSize = get_local_size(0);
   unsigned int groupID = get_group_id(0);
   unsigned int localID = get_local_id(0);
   kNode splitNode, left, right;
   int privateLeafFlag = -1, smallNodesTreated = 0, totalSmallNodes = 0;
   __local kNode splitNodeLocal;
   __local int leafCounter;
   __local int lastProcessedFlag;

   // Int local array to perform parallel scans
   __local int localIntArray[WORKGROUPSIZE];
   // GPU_REAL local array to perform parallel reductions
   __local GPU_REAL localGpuRealArray[WORKGROUPSIZE];

   // Queue in local memory to store small nodes
   __local kNode smallSplitsQueue[WORKGROUPSIZE];
   // Queue in local memory to store small nodes' indexes in the global work queue
   __local int smallSplitsQueueIndexes[WORKGROUPSIZE];

   if(get_global_id(0) == 0)
      smallNodesCounter[0] = 0;
   if(localID == 0)
      leafCounter = 0;

   // Work-Group Splits
   int dequeueIndex = groupID;
```

```
while(dequeueIndex < queueSize) {
   if(localID == 0) // Read a split object from the input queue
      splitNodeLocal = inputQueue[dequeueIndex];
   barrier(CLK_LOCAL_MEM_FENCE);
   splitNode = splitNodeLocal;

   int beginTri = splitNode.beginTri;
   int endTri = splitNode.endTri;
   int numTriangles = endTri - beginTri + 1;

   // If the node is "small", put it on a local queue to be
   // processed individually by a single thread
   if(numTriangles <= SMALL_SPLIT_THRESHOLD && smallNodesTreated < 128) {
      if(localID == 0) {
         smallSplitsQueue[smallNodesTreated] = splitNode;
         smallSplitsQueueIndexes[smallNodesTreated] = dequeueIndex;
      }
      smallNodesTreated++;
      totalSmallNodes++;
   }
   else {
      // When the node is a small node but there is no more local
      // memory to store it, we simply increment the small nodes counter
      if(numTriangles <= SMALL_SPLIT_THRESHOLD)
         totalSmallNodes++;
      // Choose the split axis by the longest direction method
      int splitDirection = getLongestDirection(splitNode.kDop);
      // Compute k-Dop split value by the mean method
      GPU_REAL overallCentroid = getOverallCentroid(beginTri, endTri, triangles,
      trianglesIndexesInput, splitDirection, localGpuRealArray);
      // Split and reorder triangles' indices
      int splitPoint = splitByCentroid(beginTri, endTri, triangles,
      trianglesIndexesInput, trianglesIndexesOutput, overallCentroid,
      splitDirection, localIntArray, &lastProcessedFlag);
      // Commit triangles' indexes to global memory to guarantee that all the threads
      // in this Work-Group read updated data during the k-Dop construction
      barrier(CLK_GLOBAL_MEM_FENCE);

      int leftNodeTriangles = splitPoint - beginTri;
      int rightNodeTriangles = (endTri+1) - splitPoint;

      // If the subdivision led to the same geometry as the current node,
      // we simply give half primitives for each node
      if((leftNodeTriangles == 0) || (rightNodeTriangles == 0))
         splitPoint = beginTri + (numTriangles/2);

      // k-Dop Construction Process
      build18Dop(&left, beginTri, splitPoint-1, trianglesIndexesOutput,
      triangles, localGpuRealArray);
      barrier(CLK_LOCAL_MEM_FENCE);

      build18Dop(&right, splitPoint, endTri, trianglesIndexesOutput,
      triangles, localGpuRealArray);
      barrier(CLK_LOCAL_MEM_FENCE);

      if(localID == 0) {
         setNodeInfo(&left, beginTri, splitPoint-1, splitNode.id,
         splitNode.leftChild, splitNode.level, leafs[1], trianglesPerNode);
         setNodeInfo(&right, splitPoint, endTri, splitNode.id,
         splitNode.rightChild, splitNode.level, leafs[1], trianglesPerNode);

         if(isLeaf(left) || isLeaf(right))
```

```
            if(privateLeafFlag < 0)
               privateLeafFlag = atom_inc(&leafCounter);
         hierarchy[splitNode.leftChild] = left;
         hierarchy[splitNode.rightChild] = right;
         outputQueue[dequeueIndex*2] = left;
         outputQueue[dequeueIndex*2+1] = right;
      }
   }
   dequeueIndex += get_num_groups(0);
}

if(localID == 0) {
   if(totalSmallNodes > 0)
      atom_add(&smallNodesCounter[0], totalSmallNodes);
}

// Individual Splits
int smallReadIndex = localID;
while(smallReadIndex < smallNodesTreated) {
   dequeueIndex = smallSplitsQueueIndexes[smallReadIndex];
   splitNode = smallSplitsQueue[smallReadIndex];
   int beginTri = splitNode.beginTri;
   int endTri = splitNode.endTri;
   int numTriangles = endTri - beginTri + 1;

   // Choose the split axis by the longest direction method
   int       splitDirection = getLongestDirection(splitNode.kDop);
   // Compute k-Dop split value by the mean method
   GPU_REAL overallCentroid = getOverallCentroidSingleThread(beginTri, endTri,
   triangles, trianglesIndexesInput, splitDirection);
   // Split and reorder triangles' indices
   int splitPoint = splitByCentroidSingleThread(beginTri, endTri, triangles,
   trianglesIndexesInput, trianglesIndexesOutput, overallCentroid, splitDirection);

   int leftTriangles = splitPoint - beginTri;
   int rightTriangles = (endTri+1) - splitPoint;

   // If the subdivision led to the same geometry as the current node,
   // we simply give half primitives for each node
   if((leftTriangles == 0) || (rightTriangles == 0))
      splitPoint = beginTri + (numTriangles/2);

   // k-Dop Construction Process
   build18DopSingleThread(&left, triangles, trianglesIndexesOutput,
   beginTri, splitPoint-1);

   build18DopSingleThread(&right, triangles, trianglesIndexesOutput,
   splitPoint, endTri);

   setNodeInfo(&left, beginTri, splitPoint-1, splitNode.id, splitNode.leftChild,
   splitNode.level, leafs[1], trianglesPerNode);
   setNodeInfo(&right, splitPoint, endTri, splitNode.id, splitNode.rightChild,
   splitNode.level, leafs[1], trianglesPerNode);

   if(isLeaf(left) || isLeaf(right))
      if(privateLeafFlag < 0)
         privateLeafFlag = atom_inc(&leafCounter);

      hierarchy[splitNode.leftChild] = left;
      hierarchy[splitNode.rightChild] = right;
      outputQueue[dequeueIndex*2] = left;
      outputQueue[dequeueIndex*2+1] = right;
```

```
         smallReadIndex += localSize;
      }

      // The first thread to find a leaf sets the leaf flag to true
      if(privateLeafFlag == 0)
      // Flag to indicate that there are leaf nodes in this level and we need
      // to perform a Compaction
         leafs[0] = 1;
}
```

A.3.4 Large Nodes Construction Kernels

A.3.4.1 Large Nodes Construction Kernels - Get Split Point

```
__kernel void getOverallCentroidGlobal(__global gpu_triangle* triangles,
__global int* trianglesIndexes, __global kNode* inputQueue, int queueSize,
__global GPU_REAL* centroidsArray, __global int* offsetsArray) {

   int localSize = get_local_size(0);
   int globalSize = get_global_size(0);
   int groupID = get_group_id(0);
   int localID = get_local_id(0);
   int numGroups = get_num_groups(0);
   int globalWriteOffset = 0;

   __local kNode splitNodeLocal;

   // GPU_REAL local array to perform parallel reductions
   __local GPU_REAL localGpuRealArray[WORKGROUPSIZE];

   // Work-Group Splits
   for(int i = 0; i < queueSize; i++) {
      if(localID == 0) // Read a split object from the input queue
      splitNodeLocal = inputQueue[i];
      barrier(CLK_LOCAL_MEM_FENCE);
      kNode splitNode = splitNodeLocal;

      int beginTri = splitNode.beginTri;
      int endTri = splitNode.endTri;
      int numTriangles = endTri - beginTri + 1;
      int groupOffset = groupID;
      int numBlocks = numTriangles/localSize;

      if(numTriangles % localSize != 0)
         numBlocks++;

      int groupReadIndex = beginTri + (groupID * localSize);

      // Choose the split axis by the longest direction method
      int splitDirection = getLongestDirection(splitNode.kDop);
      while(groupReadIndex <= endTri) {
         int lastIndex = min(groupReadIndex+localSize-1, endTri);
         // Compute k-Dop split value by the mean method
         GPU_REAL overallCentroid = getOverallCentroid(groupReadIndex, lastIndex,
         triangles, trianglesIndexes, splitDirection, localGpuRealArray);
         if(localID == 0) {
            int index = globalWriteOffset + groupOffset;
            centroidsArray[index] = overallCentroid;
         }
         groupReadIndex += globalSize;
         groupOffset += numGroups;
      }
      if(groupID == 0 && localID == 0) {
         offsetsArray[i*2] = globalWriteOffset;
         offsetsArray[i*2+1] = globalWriteOffset+numBlocks-1;
      }
      globalWriteOffset += numBlocks;
   }
}

__kernel void mergeCentroids(__global GPU_REAL* centroidsArray,
__global int* offsetsArray, __global kNode* inputQueue, int queueSize,
__global GPU_REAL* outputCentroids) {

   __local GPU_REAL reductionArray[WORKGROUPSIZE];
   unsigned int groupID = get_group_id(0);
   unsigned int localSize = get_local_size(0);
   unsigned int localID = get_local_id(0);
   unsigned int numGroups = get_num_groups(0);
   GPU_REAL acum;
   int groupReadIndex = groupID;

   while(groupReadIndex < queueSize) {
      int startIdx = offsetsArray[groupReadIndex*2]; // where the blocks begin
      int endIdx = offsetsArray[groupReadIndex*2+1];
      int readIndex = startIdx + localID;
      acum = 0;
      while(readIndex <= endIdx) {
         acum += centroidsArray[readIndex];
         readIndex += localSize;
      }
      reductionArray[localID] = acum;
      barrier(CLK_LOCAL_MEM_FENCE);
      // Reduction Algorithm
      GPU_REAL sum = reduceSum(reductionArray, localSize);
      if(localID == 0) {
         kNode node = inputQueue[groupReadIndex];
         int beginTri = node.beginTri;
         int endTri = node.endTri;
         int numTriangles = endTri - beginTri + 1;
         GPU_REAL numCentroids = numTriangles / (GPU_REAL)localSize;
         outputCentroids[groupReadIndex] = sum / numCentroids;
      }
      groupReadIndex += numGroups;
   }
}
```

A.3.4.2 Large Nodes Construction Kernels - Split By Centroid

```
__kernel void splitByCentroidGlobal(__global gpu_triangle* triangles,
__global int* trianglesIndexesInput, __global GPU_REAL* centroids,
__global kNode* inputQueue, __global int* scannedArray,
__global int* unScannedArray, __global int* blockSums, int queueSize) {

   unsigned int groupID = get_group_id(0);
   unsigned int localSize = get_local_size(0);
   unsigned int globalSize = get_global_size(0);
   unsigned int localID = get_local_id(0);
   unsigned int numGroups = get_num_groups(0);
   __local kNode splitNodeLocal;
   __local GPU_REAL centroidLocal;
   __local int scanArray[WORKGROUPSIZE];
   int blockSumsWriteIndex = 0, flag;
   for(int i = 0; i < queueSize; i++) {
      if(localID == 0) { // Read a split object from the input queue
         splitNodeLocal = inputQueue[i];
         centroidLocal = centroids[i];
      }
      barrier(CLK_LOCAL_MEM_FENCE);
      kNode splitNode = splitNodeLocal;

      int beginTri = splitNode.beginTri;
      int endTri = splitNode.endTri;
      int numTriangles = endTri - beginTri + 1;

      int numBlockSums = numTriangles / localSize;
      if(numTriangles % localSize != 0)
         numBlockSums++;

      int splitDirection = getLongestDirection(splitNode.kDop);
      int groupOffset = beginTri + (groupID*localSize);
      int block = 0;
      while(groupOffset < endTri) {
         int readIndex = groupOffset + localID;
         if(readIndex <= endTri) {
            int triangleIndex = trianglesIndexesInput[readIndex];
            gpu_triangle triangle = triangles[triangleIndex];
            GPU_REAL centroid = triangleCentroid(splitDirection, triangle);
            if(centroid < centroidLocal)
               flag = 1;
            else
               flag = 0;
         }
         else
            flag = 0;

         int bankOffset = CONFLICT_FREE_OFFSET(localID);
         scanArray[localID + bankOffset] = flag;
         // Scan Algorithm
         scan(scanArray, localSize);
         barrier(CLK_LOCAL_MEM_FENCE);
         if(readIndex <= endTri) {
            unScannedArray[readIndex] = flag;
            scannedArray[readIndex] = scanArray[localID + bankOffset];
         }
         if(localID == localSize-1) {
            int index = blockSumsWriteIndex + groupID + (numGroups*block);
            blockSums[index] = scanArray[localID + bankOffset] + flag;
         }
         block++;
         groupOffset += globalSize;
      }
      blockSumsWriteIndex += numBlockSums;
   }
}

__kernel void mergeSplitByCentroid(__global gpu_triangle* triangles,
__global int* trianglesIndexesInput, __global int* trianglesIndexesOutput,
__global kNode* inputQueue, __global int* scannedArray, int queueSize,
__global int* unScannedArray, __global int* blockSums, __global int* splitPoints) {

   unsigned int groupID = get_group_id(0);
   int localSize = get_local_size(0);
   unsigned int globalSize = get_global_size(0);
   unsigned int localID = get_local_id(0);
   unsigned int numGroups = get_num_groups(0);
   __local kNode splitNodeLocal;
   __local int scanArray[WORKGROUPSIZE];
   int newTriPos, readIndex;
   int blockSumsReadIndex = 0, flag, localIncrement;
   int blockIncrement, beginTri, endTri, numTriangles, numBlockSums;

   for(int i = 0; i < queueSize; i++) {
      if(localID == 0) // Read a split object from the input queue
         splitNodeLocal = inputQueue[i];

      barrier(CLK_LOCAL_MEM_FENCE);
      kNode splitNode = splitNodeLocal;

      localIncrement = 0;
      beginTri = splitNode.beginTri;
      endTri = splitNode.endTri;
      numTriangles = endTri - beginTri + 1;
      numBlockSums = numTriangles / localSize;
      if(numTriangles % localSize != 0)
         numBlockSums++;

      // Scan the Block Sums Array to get the group increment
      int bankOffset = CONFLICT_FREE_OFFSET(localID);
      int globalRIdx = groupID;
      int localRIdx = groupID;

      int block = 1;
      // scan block sums in blocks with the work group size
      while(globalRIdx < numBlockSums) {
         readIndex = blockSumsReadIndex + ((block-1)*localSize) + localID;
         if(readIndex < (blockSumsReadIndex + numBlockSums))
            scanArray[localID + bankOffset] = blockSums[readIndex];
         else
            scanArray[localID + bankOffset] = 0;
         // Scan Algorithm
         scan(scanArray, localSize);
         barrier(CLK_LOCAL_MEM_FENCE);
         // get several group block increments and reorder primitives
         int blockReadOffset = min(localSize*block, numBlockSums);
         while(globalRIdx < blockReadOffset) {
            localRIdx = globalRIdx % localSize;
            int groupBankOffset = CONFLICT_FREE_OFFSET(localRIdx);
            blockIncrement = scanArray[localRIdx + groupBankOffset];

            // Sorting Process
            int negatives=(globalRIdx*localSize)-blockIncrement-localIncrement;
            int thrIdx = beginTri + localID + (globalRIdx*localSize);
            if(thrIdx <= endTri) {
               int triIdxIn = trianglesIndexesInput[thrIdx];
               flag = unScannedArray[thrIdx];
               if(flag == 1) // Triangles that will be placed in the left child
                  newTriPos=beginTri+scannedArray[thrIdx]+blockIncrement+localIncrement;
               else // Triangles that will be placed in the right child
                  newTriPos = (endTri - negatives) - (localID - scannedArray[thrIdx]);
               // Saving indexes' new order
               trianglesIndexesOutput[newTriPos] = triIdxIn;
            }
         globalRIdx += numGroups;
         }
         int bankOffsetLastThread = CONFLICT_FREE_OFFSET(localSize-1);
```

```
         int lastProcessedIndex = min(blockSumsReadIndex+(block*localSize)-1,
         blockSumsReadIndex+numBlockSums-1);

         if(blockSumsReadIndex+(block*localSize)-1 < blockSumsReadIndex+numBlockSums)
            localIncrement += scanArray[(localSize-1) + bankOffsetLastThread] +
            blockSums[lastProcessedIndex];
         else
            localIncrement += scanArray[(localSize-1) + bankOffsetLastThread];
         block++;
         barrier(CLK_LOCAL_MEM_FENCE);
      }
      blockSumsReadIndex += numBlockSums;
      if(groupID == 0 && localID == 0)
         splitPoints[i] = beginTri + localIncrement;
   }
}
```

A.3.4.3 Large Nodes Construction Kernels - Build k-Dops

```
__kernel void buildDops(__global gpu_triangle* triangles,
__global int* trianglesIndexes, __global kNode* outputDops,
__global kNode* inputQueue, __global int* splitPoints,
int queueSize, int nTriangles) {

   unsigned int localSize = get_local_size(0);
   unsigned int localID = get_local_id(0);
   unsigned int globalSize = get_global_size(0);
   unsigned int groupID = get_group_id(0);
   unsigned int numGroups = get_num_groups(0);
   int beginTri, endTri;
   int numDops = 0;
   // GPU_REAL local array to perform parallel reductions
   __local GPU_REAL localGpuRealArray[WORKGROUPSIZE];
   if(queueSize == 0) { // Root Building!
      // k-Dop Construction Process executed by several Work-Groups
      int groupReadOffset = localSize * get_group_id(0);
      int lastTriIndex;
      int j = 0;
      while(groupReadOffset < nTriangles) {
         kNode node;
         // Build the root's k-Dop
         lastTriIndex = groupReadOffset+localSize-1;
         if(lastTriIndex >= nTriangles)
            lastTriIndex = nTriangles-1;
         build18Dop(&node, groupReadOffset, lastTriIndex,
         trianglesIndexes, triangles, localGpuRealArray);
         if(localID == 0) {
            int writeIndex = groupID + (numGroups*j);
            outputDops[writeIndex] = node;
         }
         barrier(CLK_LOCAL_MEM_FENCE);
         groupReadOffset += globalSize;
         j++;
      }
   }
   else {
      for(int i = 0; i < queueSize; i++) {
         kNode inputNode = inputQueue[i];
         beginTri = inputNode.beginTri;
         kNode node;
         endTri = splitPoints[i]-1;
         for(int n = 0; n < 2; n++) { // build 2 nodes
```

```
            int groupReadOffset = beginTri + (localSize * get_group_id(0));
            int lastTriIndex;
            int j = 0;
            while(groupReadOffset <= endTri) {
               lastTriIndex = groupReadOffset+localSize-1;
               if(lastTriIndex > endTri)
                  lastTriIndex = endTri;
               build18Dop(&node, groupReadOffset, lastTriIndex,
               trianglesIndexes, triangles, localGpuRealArray);
               if(localID == 0) {
                  int writeIndex = numDops + groupID + (numGroups*j);
                  node.id = groupReadOffset;
                  outputDops[writeIndex] = node;
               }
               barrier(CLK_LOCAL_MEM_FENCE);
               groupReadOffset += globalSize;
               j++;
            }
            // change info for the right node construction
            numDops += (endTri - beginTri + 1) / localSize;
            if((endTri - beginTri + 1) % localSize != 0)
            numDops++;
            beginTri = endTri+1;
            endTri = inputNode.endTri;
         }
      }
   }
}

/**
 * Merge Root Dops Kernel, without reading nodes from input queue (for simplification)
**/
__kernel void mergeRootDops(__global kNode* inputDops, __global kNode* outputNodes,
__global kNode* hierarchy, int nTriangles, int trianglesPerNode) {

   GPU_REAL k[KDOP_SIZE];
   kNode kNodeDop;
   kNode node;
   __local GPU_REAL reductionArray[WORKGROUPSIZE*2];

   unsigned int localID = get_local_id(0);
   unsigned int localSize = get_local_size(0);

   int nDops = nTriangles / WORKGROUPSIZE;
   if(nTriangles % WORKGROUPSIZE != 0)
      nDops++;
   int dopsPerThread = nDops / localSize;
   if(nDops % localSize != 0)
      dopsPerThread++;

   for(int t = 0; t < dopsPerThread; t++) {
      int index = localID + (localSize*t);
      if(index >= nDops)
         index = nDops-1;
      kNodeDop = inputDops[index];
      for(int i=0, j=HALF_KDOP_SIZE;
      i < HALF_KDOP_SIZE && j < KDOP_SIZE; i++,j++) {
         reductionArray[localID] = kNodeDop.kDop[i];
         barrier(CLK_LOCAL_MEM_FENCE);
         k[i] = reduceMinMax(reductionArray, DESCENDING, localSize);
         reductionArray[localID] = kNodeDop.kDop[j];
         barrier(CLK_LOCAL_MEM_FENCE);
         k[j] = reduceMinMax(reductionArray, ASCENDING, localSize);
      }
      // Thread 0 updates the k-Dop with the current values
      if(localID == 0) {
         if(t == 0) {
            for(int i=0, j=HALF_KDOP_SIZE;
            i < HALF_KDOP_SIZE && j < KDOP_SIZE; i++,j++) {
               node.kDop[i] = k[i];
               node.kDop[j] = k[j];
            }
         }
         else {
            for(int i=0, j=HALF_KDOP_SIZE; i<HALF_KDOP_SIZE && j<KDOP_SIZE;i++,j++) {
               node.kDop[i] = fmin(k[i], node.kDop[i]);
               node.kDop[j] = fmax(k[j], node.kDop[j]);
            }
         }
      }
      barrier(CLK_LOCAL_MEM_FENCE);
   }
   if(localID == 0) {
      setNodeInfo(&node, 0, nTriangles-1, -1, 0, -1, 0, trianglesPerNode);
      outputNodes[0] = node;
      hierarchy[0] = node;
   }
}

__kernel void mergeDops(__global kNode* inputDops, __global kNode* outputNodes,
__global kNode* inputQueue, __global kNode* hierarchy,
__global int* splitPoints, int queueSize, int trianglesPerNode) {

   GPU_REAL k[KDOP_SIZE];
   kNode kNodeDop;
   kNode inputNode, node;

   unsigned int localID = get_local_id(0);
   unsigned int localSize = get_local_size(0);

   __local GPU_REAL reductionArray[WORKGROUPSIZE*2];
   int beginTri, endTri, nTriangles, nDops, readOffset = 0, writeIndex = 0;

   for(int inode = 0; inode < queueSize; inode++) {
      inputNode = inputQueue[inode];
      beginTri = inputNode.beginTri;
      endTri = splitPoints[inode]-1;
      nTriangles = endTri - beginTri + 1;

      for(int child = 0; child < 2; child++) {
         nDops = nTriangles/WORKGROUPSIZE;
         if(nTriangles%WORKGROUPSIZE != 0)
            nDops++;
         int dopsPerThread = nDops / localSize;
         if(nDops % localSize != 0)
            dopsPerThread++;

         for(int t = 0; t < dopsPerThread; t++) {
            int index = readOffset + localID + (localSize*t);
            if(index >= readOffset+nDops)
               index = readOffset+nDops-1;
            kNodeDop = inputDops[index];
            for(int i=0, j=HALF_KDOP_SIZE;
            i < HALF_KDOP_SIZE && j < KDOP_SIZE;i++,j++) {
               reductionArray[localID] = kNodeDop.kDop[i];
               barrier(CLK_LOCAL_MEM_FENCE);
               k[i] = reduceMinMax(reductionArray, DESCENDING, localSize);
               reductionArray[localID] = kNodeDop.kDop[j];
               barrier(CLK_LOCAL_MEM_FENCE);
               k[j] = reduceMinMax(reductionArray, ASCENDING, localSize);
            }
            // Thread 0 updates the k-Dop with the current values
            if(localID == 0) {
               if(t == 0) {
                  for(int i=0, j=HALF_KDOP_SIZE;
                  i < HALF_KDOP_SIZE && j < KDOP_SIZE;i++,j++) {
                     node.kDop[i] = k[i];
                     node.kDop[j] = k[j];
                  }
               }
               else {
                  for(int i=0, j=HALF_KDOP_SIZE;
                  i < HALF_KDOP_SIZE && j < KDOP_SIZE;i++,j++) {
                     node.kDop[i] = fmin(k[i], node.kDop[i]);
                     node.kDop[j] = fmax(k[j], node.kDop[j]);
                  }
               }
            }
            barrier(CLK_LOCAL_MEM_FENCE);
         }
         if(localID == 0) {
            int nodeID;
            if(child == 0)
               nodeID = inputNode.leftChild;
            else
               nodeID = inputNode.rightChild;
            setNodeInfo(&node, beginTri, endTri, inputNode.id, nodeID,
            inputNode.level, 0, trianglesPerNode);
            outputNodes[inode*2+child] = node;
            hierarchy[nodeID] = node;
         }
         readOffset += nDops;
         beginTri = endTri+1;
         endTri = inputNode.endTri;
         nTriangles = endTri - beginTri + 1;
      }
   }
}
```

A.3.5 Compaction Kernels

A.3.5.1 Compaction Kernel for Small Arrays

```
/*
 * Compaction kernel for arrays up to twice the work group size
 */
__kernel void compactionSmallArrays(__global kNode* hierarchy,
__global kNode* inputQueue, __global kNode* outputQueue,
__global int* leafs, int queue_size) {

   kNode nodeA, nodeB;
   bool isLeafA, isLeafB;
   __local int scanArray[WORKGROUPSIZE*2];

   int indexA, indexB;
   int local_id = get_local_id(0);
   int local_size = get_local_size(0);
   int n = nextPow2(queue_size);

   int ai = local_id;
   int bi = local_id + (n/2);
   int bankOffsetA = CONFLICT_FREE_OFFSET(ai);
   int bankOffsetB = CONFLICT_FREE_OFFSET(bi);

   if(bi < n) {
      nodeA = inputQueue[ai];
      isLeafA = ((nodeA.leftChild == -1) && (nodeA.rightChild == -1));
      scanArray[ai + bankOffsetA] = !isLeafA;
      if(bi < queue_size) {
         nodeB = inputQueue[bi];
         isLeafB = ((nodeB.leftChild == -1) && (nodeB.rightChild == -1));
      }
      else
         isLeafB = true;
      scanArray[bi + bankOffsetB] = !isLeafB;

      // Scan Algorithm
      scan(scanArray, n);
      barrier(CLK_LOCAL_MEM_FENCE);

      // Sorting Phase
      if(bi < queue_size) {
         if(!isLeafA) {
            indexA = scanArray[ai + bankOffsetA];
            int previous_indexA = nodeA.id;
            int new_indexA = nodeA.id - ai + indexA;
            nodeA.leftChild = (new_indexA-leafs[1]) * 2 + 1;
            nodeA.rightChild = (new_indexA-leafs[1]) * 2 + 2;
            hierarchy[previous_indexA] = nodeA;
            outputQueue[indexA] = nodeA;
         }
         if(!isLeafB) {
            indexB = scanArray[bi + bankOffsetB];
            int previous_indexB = nodeB.id;
            int new_indexB = nodeB.id - bi + indexB;
            nodeB.leftChild = (new_indexB-leafs[1]) * 2 + 1;
            nodeB.rightChild = (new_indexB-leafs[1]) * 2 + 2;
            hierarchy[previous_indexB] = nodeB;
            outputQueue[indexB] = nodeB;
         }
         else {
            if(bi == queue_size-1)
               indexB = scanArray[bi + bankOffsetB];
         }
      }
      if(bi == queue_size-1) {
         leafs[1] += queue_size - (indexB + !isLeafB); // increment leafs counter
         leafs[0] = 0; // reset flag
      }
   }
}
```

A.3.5.2 Compaction Kernels for Large Arrays

```
/**
 * Compaction kernel for Large Arrays - Phase 1
 */
```

```
__kernel void compactionLargeArrays1(__global kNode* inputQueue,
__global int* block_increments, __global int* scanned_elements, int queue_size) {

   int local_id = get_local_id(0);
   int local_size = get_local_size(0);
   int groupID = get_group_id(0);
   bool isLeaf;
   __local int scanArray[WORKGROUPSIZE*2];

   int node_index = local_id + (groupID * local_size);
   int bankOffset = CONFLICT_FREE_OFFSET(local_id);
   if(node_index < queue_size) {
      kNode node = inputQueue[node_index];
      isLeaf = ((node.leftChild == -1) && (node.rightChild == -1));
   }
   else
      isLeaf = true;

   scanArray[local_id + bankOffset] = !isLeaf;
   scan(scanArray, local_size);
   barrier(CLK_LOCAL_MEM_FENCE);
   scanned_elements[node_index] = scanArray[local_id + bankOffset];

   if(local_id == local_size-1)
      block_increments[groupID] = scanArray[local_id + bankOffset] + !isLeaf;
}

/**
 * Compaction kernel for Large Arrays - Phase 2
 */
__kernel void compactionLargeArrays2(__global int* block_increments,
__global int* block_sums, int block_sums_size) {

   int local_id = get_local_id(0);
   int local_size = get_local_size(0);
   int groupID = get_group_id(0);
   int val = 0;
   __local int scanArray[WORKGROUPSIZE*2];

   int node_index = local_id + (groupID * local_size);
   int bankOffset = CONFLICT_FREE_OFFSET(local_id);
   if(node_index < block_sums_size)
   val = block_sums[node_index];

   scanArray[local_id + bankOffset] = val;
   scan(scanArray, local_size);
   barrier(CLK_LOCAL_MEM_FENCE);
   block_sums[node_index] = scanArray[local_id + bankOffset];

   if(local_id == local_size-1)
      block_increments[groupID] = scanArray[local_id + bankOffset] + val;
}

/**
 * Compaction kernel for Large Arrays - Phase 3
 */
__kernel void compactionLargeArrays3(__global kNode* hierarchy,
__global kNode* inputQueue, __global kNode* outputQueue,
__global int* leafs, __global int* block_sums, __global int* scanned_elements,
int queue_size, int blocks, int current_leafs, __global int* block_increments,
int block_increments_size) {
   int local_id = get_local_id(0);
   int local_size = get_local_size(0);
   int groupID = get_group_id(0);
   __local int scanArray[WORKGROUPSIZE*2];

   int leafsC = current_leafs;
   int node_index = local_id + (groupID * local_size);

   if(block_increments_size > 1) {
      int bankOffset = CONFLICT_FREE_OFFSET(local_id);
      int n = nextPow2(block_increments_size);
      if(local_id < block_increments_size)
         scanArray[local_id + bankOffset] = block_increments[local_id];
      else
         scanArray[local_id + bankOffset] = 0;

      scan(scanArray, local_size);
      barrier(CLK_LOCAL_MEM_FENCE);
   }

   int increment2 = 0;
   if(groupID >= WORKGROUPSIZE*2) {
      int pos = groupID / (WORKGROUPSIZE*2);
      int bankOffset = CONFLICT_FREE_OFFSET(pos);
      increment2 = scanArray[pos + bankOffset];
   }
   int increment = block_sums[groupID] + increment2;

   // Sorting Phase
   if(node_index < queue_size) {
      kNode node = inputQueue[node_index];
      bool isLeaf = ((node.leftChild == -1) && (node.rightChild == -1));
      if(!isLeaf) {
         int index = scanned_elements[node_index] + increment;
         int previous_index = node.id;
         int new_index = node.id - node_index + index;
         int diff = previous_index - new_index;
         node.leftChild = (new_index-leafsC) * 2 + 1;
         node.rightChild = (new_index-leafsC) * 2 + 2;
         hierarchy[previous_index] = node;
         outputQueue[index] = node;
      }
   }
   if((groupID == get_num_groups(0)-1) && (local_id == local_size-1)) {
      increment leafs counter
      if(block_increments_size <= 1)
         leafs[1] += queue_size - block_increments[block_increments_size-1];
      else {
         int bankOffset = CONFLICT_FREE_OFFSET(block_increments_size);
         leafs[1] += queue_size - scanArray[block_increments_size + bankOffset];
      }
      leafs[0] = 0; // reset flag
   }
}
```

A.4 Broadphase Collision Detection Kernels

A.4.1 Simple Broadphase Collision Detection Kernel

```
/**
```

```
 * Broadphase Collision detection kernel
 */
__kernel void intersectNodes(__global int* collisionsA, __global int* collisionsB,
__global kNode* bvhA, __global kNode* bvhB, __global int* cloth_tri_indices,
__global int* object_tri_indices, __global int* numTriangles, __global int* numCollisions,
__global int* numTests, int clothSize, int objectSize) {

   __local int writeIndex;
   __local int readIndex;
   __local int read_idx_aux;
   __local int write_idx_aux;
   __local int trianglePairs;
   __local int collisions;
   __local int cloth_overlaping[MAX_SIZE2];
   __local int object_overlaping[MAX_SIZE2];

   int localID = get_local_id(0);
   int groupID = get_group_id(0);

   // Thread 0 processes both BVH roots
   if(localID == 0){
      writeIndex = 0;
      readIndex = 0;
      read_idx_aux = 0;
      write_idx_aux = 0;
      trianglePairs = 0;
      collisions = 0;

      kNode nodeA = bvhA[0];
      kNode nodeB = bvhB[0];

      if(overlap(nodeA.kDop, nodeB.kDop)) {
         cloth_overlaping[0] = nodeA.leftChild;
         object_overlaping[0] = nodeB.leftChild;
         cloth_overlaping[1] = nodeA.leftChild;
         object_overlaping[1] = nodeB.rightChild;
         cloth_overlaping[2] = nodeA.rightChild;
         object_overlaping[2] = nodeB.leftChild;
         cloth_overlaping[3] = nodeA.rightChild;
         object_overlaping[3] = nodeB.rightChild;
         write_idx_aux = 4;
         writeIndex = 4;
      }
   }
   barrier(CLK_LOCAL_MEM_FENCE);

   // Work-Group 0 does the entire tree traversal
   if(groupID == 0) {
      while(readIndex < writeIndex) {
         if(localID < write_idx_aux - read_idx_aux) {
            int r = atom_inc(&readIndex);
            int ci = cloth_overlaping[r%MAX_SIZE2];
            int oi = object_overlaping[r%MAX_SIZE2];
            kNode nodeA = bvhA[ci];
            kNode nodeB = bvhB[oi];
            if(overlap(nodeA.kDop, nodeB.kDop)) {
               // If both nodes are leafs, we save their pairing triangles' indexes
               if(isLeaf(nodeA) && isLeaf(nodeB)) {
                  int nodeANumTri = nodeA.endTri - nodeA.beginTri + 1;
                  int nodeBNumTri = nodeB.endTri - nodeB.beginTri + 1;
                  int totalPairs = nodeANumTri * nodeBNumTri;
                  int triPairsIndex = atom_add(&trianglePairs, totalPairs);
```

```
                int col = atom_inc(&collisions);
                int pos = 0;
                for(int i = 0; i < nodeANumTri; i++)
                   for(int j = 0; j < nodeBNumTri; j++) {
                      collisionsA[triPairsIndex]=cloth_tri_indices[nodeA.beginTri+i];
                      collisionsB[triPairsIndex]=object_tri_indices[nodeB.beginTri+j];
                      triPairsIndex++;
                   }
             }
             else if(isLeaf(nodeA) { // && !isLeaf(nodeB)
                int i = atom_add(&writeIndex, 2);
                cloth_overlaping[i%MAX_SIZE2] = nodeA.id;
                object_overlaping[i%MAX_SIZE2] = nodeB.leftChild;
                cloth_overlaping[(i+1)%MAX_SIZE2] = nodeA.id;
                object_overlaping[(i+1)%MAX_SIZE2] = nodeB.rightChild;
             }
             else if(isLeaf(nodeB) { // && !isLeaf(nodeA)
                int i = atom_add(&writeIndex, 2);
                cloth_overlaping[i%MAX_SIZE2] = nodeA.leftChild;
                object_overlaping[i%MAX_SIZE2] = nodeB.id;
                cloth_overlaping[(i+1)%MAX_SIZE2] = nodeA.rightChild;
                object_overlaping[(i+1)%MAX_SIZE2] = nodeB.id;
             }
             else {
                int i = atom_add(&writeIndex, 4);
                cloth_overlaping[i%MAX_SIZE2] = nodeA.leftChild;
                object_overlaping[i%MAX_SIZE2] = nodeB.leftChild;
                cloth_overlaping[(i+1)%MAX_SIZE2] = nodeA.leftChild;
                object_overlaping[(i+1)%MAX_SIZE2] = nodeB.rightChild;
                cloth_overlaping[(i+2)%MAX_SIZE2] = nodeA.rightChild;
                object_overlaping[(i+2)%MAX_SIZE2] = nodeB.leftChild;
                cloth_overlaping[(i+3)%MAX_SIZE2] = nodeA.rightChild;
                object_overlaping[(i+3)%MAX_SIZE2] = nodeB.rightChild;
             }
          }
       }
       barrier(CLK_LOCAL_MEM_FENCE);
       if(localID == 0) {
          atom_xchg(&read_idx_aux, readIndex);
          atom_xchg(&write_idx_aux, writeIndex);
       }
       barrier(CLK_LOCAL_MEM_FENCE);
    }
    if(localID == 0) {
       numTests[0] = writeIndex - readIndex;//writeIndex + 1;
       numCollisions[0] = collisions;
       numTriangles[0] = trianglePairs;
    }
 }
}
```

A.4.2 Front-Based Broadphase Collision Detection Kernel

```
/**
 * Broadphase Collision Detection with front
 */
__kernel void intersectNodesFront(__global int* outputFront1,
__global int* outputFront2, __global int* inputFront1,
__global int* inputFront2, int frontSize, __global int* new_front_size,
__global int* collisions1, __global int* collisions2, __global kNode* bvhA,
__global kNode* bvhB, __global int* cloth_tri_indices, __global int* object_tri_indices,
__global int* num_collisions, __global int* num_tests, __global int* num_triangles,
int clothSize, int objectSize) {

   __local int writeIndex;
   __local int readIndex;
   __local int read_idx_aux;
   __local int write_idx_aux;
   __local int cloth_processing_queue[MAX_SIZE];
   __local int object_processing_queue[MAX_SIZE];
   // number of triangle pairs to process on the Narrowphase
   __local int triangles_ctr;
   // front size in each group
   __local int local_front_size;
   // number of nodes tested during front reading and front climbing
   __local int nodes_processed;
   // number of colliding leaf nodes
   __local int nodes_colliding;

   int offset = 5000 * get_group_id(0);
   int offset_2 = 20000 * get_group_id(0);
   // distribute the same number of nodes per group
   int pairsPerGroup = frontSize / get_num_groups(0);
   // index where each group starts reading the front nodes to process
   int group_read_offset = pairsPerGroup * get_group_id(0);
   if(get_group_id(0) == get_num_groups(0)-1) // last group reads the remaining nodes
      pairsPerGroup += frontSize % get_num_groups(0);

   int pairsPerThread = (pairsPerGroup) / get_local_size(0);
   if((pairsPerGroup) % get_local_size(0) != 0)
      pairsPerThread++;

   if(get_local_id(0) == 0) {
      writeIndex = 0;
      readIndex = 0;
      read_idx_aux = 0;
      write_idx_aux = 0;
      triangles_ctr = 0;
      local_front_size = 0;
      nodes_colliding = 0;
      nodes_processed = 0;
   }
   barrier(CLK_LOCAL_MEM_FENCE);

   for(int pair = 0; pair < pairsPerThread; pair++) {
      int thread_read_offset = get_local_id(0) + pair * get_local_size(0);
      int pairID = thread_read_offset + group_read_offset;
      if(thread_read_offset < pairsPerGroup && pairID < frontSize) {
         int c_node_idx = inputFront1[pairID];
         int o_node_idx = inputFront2[pairID];
         kNode nodeA = bvhA[c_node_idx];
         kNode nodeB = bvhB[o_node_idx];
         bool finish = false;
         while(!finish) {
            if(overlap(nodeA.kDop, nodeB.kDop)) { // nodes overlap
               int i = atom_inc(&writeIndex);
               cloth_processing_queue[i] = nodeA.id;
               object_processing_queue[i] = nodeB.id;
               finish = true;
            }
            else { // pair do not overlap: climbing the tree...
               if((nodeA.parent<0) && (nodeB.parent<0)){//we have reached the root
                  outputFront1[0] = nodeA.id;
                  outputFront2[0] = nodeB.id;
```

```
                 local_front_size = 1; // only one thread can reach the root
                 finish = true;
              }
              // choose BVH A node to climb
              else if((nodeA.level > nodeB.level) && (nodeA.parent >= 0)) {
                 kNode parent = bvhA[nodeA.parent];
                 // top level nodes overlap, we add the children to the front
                 if(overlap(parent.kDop, nodeB.kDop)) {
                    int c = atom_inc(&local_front_size);
                    outputFront1[c + offset] = nodeA.id;
                    outputFront2[c + offset] = nodeB.id;
                    finish = true;
                 }
                 else { // top level do not overlap, keep climbing
                    if(nodeA.id == parent.leftChild)
                       nodeA = parent;
                    else
                       finish = true;
                 }
              }
              else { // choose BVH B node to climb
                 kNode parent = bvhB[nodeB.parent];
                 if(overlap(nodeA.kDop, parent.kDop)) {
                    int c = atom_inc(&local_front_size);
                    outputFront1[c + offset] = nodeA.id;
                    outputFront2[c + offset] = nodeB.id;
                    finish = true;
                 }
                 else { // top level do not overlap, keep climbing
                    if(nodeB.id == parent.leftChild)
                       nodeB = parent;
                    else
                       finish = true;
                 }
              }
           }
           atom_inc(&nodes_processed);
        }
     }
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  if(get_local_id(0) == 0)
     write_idx_aux = writeIndex;
  barrier(CLK_LOCAL_MEM_FENCE);

  while(readIndex < writeIndex) {
     if(get_local_id(0) < write_idx_aux - read_idx_aux) {
        int r = atom_inc(&readIndex);
        int cIdx = cloth_processing_queue[r%MAX_SIZE];
        int oIdx = object_processing_queue[r%MAX_SIZE];
        kNode nodeA = bvhA[cIdx];
        kNode nodeB = bvhB[oIdx];
        if(overlap(nodeA.kDop, nodeB.kDop)) {
           if(isLeaf(nodeA) && isLeaf(nodeB)) {
              int c_leaf_triangles = nodeA.endTri - nodeA.beginTri + 1;
              int o_leaf_triangles = nodeB.endTri - nodeB.beginTri + 1;
              int n_tests = c_leaf_triangles * o_leaf_triangles;
              int rc = atom_add(&triangles_ctr, n_tests);
              int x = 0;
              for(int i = 0; i < c_leaf_triangles; i++) {
                 for(int j = 0; j < o_leaf_triangles; j++) {
                    collisions1[offset_2+rc+x]=cloth_tri_indices[nodeA.beginTri+i];
```

```
                  collisions2[offset_2+rc+x]=object_tri_indices[nodeB.beginTri+j];
                  x++;
               }
            }
            int c = atom_inc(&local_front_size);
            outputFront1[c + offset] = nodeA.id;
            outputFront2[c + offset] = nodeB.id;
            atom_inc(&nodes_colliding);
         }
         else {
            int i = atom_add(&writeIndex, 2);
            if(nodeA.level <= nodeB.level && !isLeaf(nodeA)) {
               cloth_processing_queue[i%MAX_SIZE] = nodeA.leftChild;
               object_processing_queue[i%MAX_SIZE] = nodeB.id;
               cloth_processing_queue[(i+1)%MAX_SIZE] = nodeA.rightChild;
               object_processing_queue[(i+1)%MAX_SIZE] = nodeB.id;
            }
            else {
               cloth_processing_queue[i%MAX_SIZE] = nodeA.id;
               object_processing_queue[i%MAX_SIZE] = nodeB.leftChild;
               cloth_processing_queue[(i+1)%MAX_SIZE] = nodeA.id;
               object_processing_queue[(i+1)%MAX_SIZE] = nodeB.rightChild;
            }
         }
      }
      else { // nodes do not overlap
         int c = atom_inc(&local_front_size);
         outputFront1[c + offset] = nodeA.id;
         outputFront2[c + offset] = nodeB.id;
      }
   }
   barrier(CLK_LOCAL_MEM_FENCE);
   if(get_local_id(0) == 0) {
      atom_xchg(&read_idx_aux, readIndex);
      atom_xchg(&write_idx_aux, writeIndex);
   }
   barrier(CLK_LOCAL_MEM_FENCE);
}
if(get_local_id(0) == 0) {
   num_tests[get_group_id(0)] = /*nodes_processed ++*/ writeIndex;
   num_collisions[get_group_id(0)] = nodes_colliding;
   num_triangles[get_group_id(0)] = triangles_ctr;
   new_front_size[get_group_id(0)] = local_front_size;
}
}

/**
 * Kernel to compact the front produced by several work groups,
 * also used to compact the intersected pairs produced by several
 * work groups
 */
__kernel void compactFront(__global int* outputFront1,
__global int* outputFront2, __global int* inputFront1,
__global int* inputFront2, __global int* frontSize, int blockSize) {

   int numNodes = 0;
   for(int i = 0; i < get_num_groups(0); i++) {
      int nodesPerThread = frontSize[i] / get_global_size(0);
      if(frontSize[i] % get_global_size(0) != 0)
         nodesPerThread++;
      for(int j = 0; j < nodesPerThread; j++) {
         int nodeID = get_global_id(0) + (get_global_size(0) * j);
         if(nodeID < frontSize[i]) {
            int node1 = inputFront1[(i * blockSize) + nodeID];
            int node2 = inputFront2[(i * blockSize) + nodeID];
            outputFront1[numNodes + nodeID] = node1;
            outputFront2[numNodes + nodeID] = node2;
         }
      }
      numNodes += frontSize[i];
   }
}
```

A.5 Narrowphase Kernel

```
/**
 * Narrowphase Collision detection kernel
 */
int compute_plane_eq(float4 v0, float4 v1, float4 v2, float4 u0,
float4 u1, float4 u2, float* du0, float* du1, float* du2, float4* n) {

   // Compute plane equation of triangle(V0,V1,V2)
   float4 e1 = v1 - v0;
   float4 e2 = v2 - v0;
   (*n) = cross(e1, e2);
   float d1 = -dot((*n), v0);

   // Put U0,U1,U2 into plane equation 1 to compute signed distances to the plane
   (*du0) = dot((*n), u0) + d1;
   (*du1) = dot((*n), u1) + d1;
   (*du2) = dot((*n), u2) + d1;

   // Coplanarity robustness check
   if(fabs((*du0)) < EPSILON_1) (*du0) = 0.0;
   if(fabs((*du1)) < EPSILON_1) (*du1) = 0.0;
   if(fabs((*du2)) < EPSILON_1) (*du2) = 0.0;

   float du0du1 = (*du0) * (*du1);
   float du0du2 = (*du0) * (*du2);

   if(du0du1 > 0.0f && du0du2 > 0.0f)
      return 0; // no intersection occurs

   return 1;
}

int compute_intervals(float vp0, float vp1, float vp2, float dv0,
float dv1, float dv2, float* isect1, float* isect2) {

   if(dv0*dv1 > 0.0f) {
      //here we know that D0D2<=0.0
      //that is D0, D1 are on the same side, D2 on the other or on the plane
      *isect1 = vp2 + (((vp0-vp2)*dv2)/(dv2-dv0));
      *isect2 = vp2 + (((vp1-vp2)*dv2)/(dv2-dv1));
      return 1;
   }
   if(dv0*dv2 > 0.0f) {
      //here we know that d0d1<=0.0
      *isect1 = vp1 + (((vp0-vp1)*dv1)/(dv1-dv0));
      *isect2 = vp1 + (((vp2-vp1)*dv1)/(dv1-dv2));
      return 1;
   }
   if(dv1*dv2 > 0.0f || dv0 != 0.0f) {
      //here we know that d0d1<=0.0 or that D0!=0.0
      *isect1 = vp0 + (((vp1-vp0)*dv0)/(dv0-dv1));
      *isect2 = vp0 + (((vp2-vp0)*dv0)/(dv0-dv2));
      return 1;
   }
   if(dv1 != 0.0f) {
      *isect1 = vp1 + (((vp0-vp1)*dv1)/(dv1-dv0));
      *isect2 = vp1 + (((vp2-vp1)*dv1)/(dv1-dv2));
      return 1;
   }
   if(dv2 != 0.0f) {
      *isect1 = vp2 + (((vp0-vp2)*dv2)/(dv2-dv0));
      *isect2 = vp2 + (((vp1-vp2)*dv2)/(dv2-dv1));
      return 1;
   }
   return 0;
}

void sort(float* a, float* b) {
   if(*a > *b) {
      float c;
      c = *a;
      *a = *b;
      *b = c;
   }
}

__kernel void tritriIntersect(__global gpu_triangle* clothTri,
__global gpu_triangle* objTri, __global int* clothIndices,
__global int* objectIndices, __global int* collTri, int num_collisions) {

   int triangles_witem = num_collisions / get_global_size(0);
   if(num_collisions % get_global_size(0) != 0)
      triangles_witem++;

   for(int i = 0; i < triangles_witem; i++) {
      int tid = get_global_id(0) + (i * get_global_size(0));
      if(tid < num_collisions) {
         int ctID = clothIndices[tid];
         int otID = objectIndices[tid];
         collTri[tid] = 1;

         // Store cloth vertices in float4 arrays
         float4 v0 = (float4)(clothTri[ctID].v0x,
         clothTri[ctID].v0y, clothTri[ctID].v0z, 0.0f);
         float4 v1 = (float4)(clothTri[ctID].v1x,
         clothTri[ctID].v1y, clothTri[ctID].v1z, 0.0f);
         float4 v2 = (float4)(clothTri[ctID].v2x,
         clothTri[ctID].v2y, clothTri[ctID].v2z, 0.0f);

         // Store objects vertices in float4 arrays
         float4 u0 = (float4)(objTri[otID].v0x,
         objTri[otID].v0y, objTri[otID].v0z, 0.0f);
         float4 u1 = (float4)(objTri[otID].v1x,
         objTri[otID].v1y, objTri[otID].v1z, 0.0f);
         float4 u2 = (float4)(objTri[otID].v2x,
         objTri[otID].v2y, objTri[otID].v2z, 0.0f);

         float4 n1, n2;
         float dv0, dv1, dv2, du0, du1, du2;
         if(compute_plane_eq(v0, v1, v2, u0, u1, u2, &du0, &du1, &du2, &n1) == 0)
            collTri[tid] = 2;
         if(compute_plane_eq(u0, u1, u2, v0, v1, v2, &dv0, &dv1, &dv2, &n2) == 0)
            collTri[tid] = 3;

         float4 d = cross(n1, n2);

         // Compute and index to the largest component of d
         float vp0, vp1, vp2, up0, up1, up2;
         float a = fabs(d.x);
         float b = fabs(d.y);
         float c = fabs(d.z);
         float max = fmax(a, b);
         max = fmax(max, c);

         // This is the simplified projection onto L
         if(b == max) {
            vp0 = v0.y; vp1 = v1.y; vp2 = v2.y;
            up0 = u0.y; up1 = u1.y; up2 = u2.y;
         }
         if(c == max) {
            vp0 = v0.z; vp1 = v1.z; vp2 = v2.z;
            up0 = u0.z; up1 = u1.z; up2 = u2.z;
         }
         else {
            vp0 = v0.x; vp1 = v1.x; vp2 = v2.x;
            up0 = u0.x; up1 = u1.x; up2 = u2.x;
         }

         // compute interval fobvhAth triangle
         float isect1_0, isect1_1;
         if(compute_intervals(vp0, vp1, vp2, dv0, dv1, dv2, &isect1_0, &isect1_1)==0)
            collTri[tid] = 4;

         // compute interval for object triangle
         float isect2_0, isect2_1;
         if(compute_intervals(up0, up1, up2, du0, du1, du2, &isect2_0, &isect2_1)==0)
            collTri[tid] = 5;

         sort(&isect1_0, &isect1_1);
         sort(&isect2_0, &isect2_1);

         if(isect1_1 < isect2_0 || isect2_1 < isect1_0)
            collTri[tid] = 6;
      }
   }
}
```

A.6 Update Kernel

```
/**
 * Merge two nodes to update their parent
 */
void merge(kNode left, kNode right, __global kNode* hierarchy) {
   int parentID = left.parent; // = right.parent
   kNode parent = hierarchy[parentID];

   for(int i=0, j=HALF_KDOP_SIZE; i<HALF_KDOP_SIZE && j<KDOP_SIZE;i++,j++) {
      parent.kDop[i] = fmin(left.kDop[i], right.kDop[i]);
      parent.kDop[j] = fmax(left.kDop[j], right.kDop[j]);
   }
   hierarchy[parentID] = parent;
   hierarchy[left.id] = left;
   hierarchy[right.id] = right;
}

/**
 * Update Kernel, invoked once per level
 */
__kernel void updateBVH(__global gpu_triangle* triangles,
__global int* tri_indices, __global kNode* hierarchy, int nTriangles,
int hierarchySize, int firstLevelNode, int levelSize, __global int* updateFlag) {

   int nodePairsPerGroup = (levelSize/2) / get_num_groups(0);
   if(get_group_id(0) == get_num_groups(0)-1)
      nodePairsPerGroup += (levelSize/2) % get_num_groups(0);

   int nodePairsPerThread = nodePairsPerGroup / get_local_size(0);
   if(get_local_id(0) < (nodePairsPerGroup % get_local_size(0)))
      nodePairsPerThread++;

   for(int i = 0; i < nodePairsPerThread; i++) {
      int node_index = firstLevelNode + (get_local_id(0)*2)+(get_local_size(0)*2*i)
      + (((levelSize/2) / get_num_groups(0)) * 2 * get_group_id(0));

      // Each thread reads two nodes (siblings), in order to update their parent
      kNode left = hierarchy[node_index];
      kNode right = hierarchy[node_index + 1];

      // If the node is a leaf, update it with its primitives
      if(isLeaf(left))
         build18DopSingleThread(&left, triangles, tri_indices,
         left.beginTri, left.endTri);
      if(isLeaf(right))
         build18DopSingleThread(&right, triangles, tri_indices,
         right.beginTri, right.endTri);

      // merge left and right child to update the parent
      merge(left, right, hierarchy);
   }
}
```

Bibliography

[1] Bullet - Project Hosting on Google Code. http://code.google.com/p/bullet.

[2] Bullet Physics Library. http://bulletphysics.org/wordpress.

[3] Havok. http://www.havok.com.

[4] NVIDIA PhysX. http://www.nvidia.com/object/physx_new.html.

[5] OpenCL Programming Guide for the CUDA Architecture. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/ OpenCL_Programming_Guide.pdf.

[6] Fernando Birra. *Técnicas Eficientes de Simulação de Tecidos com Realismo Acrescido*. Tese de doutouramento, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, 2007.

[7] Christian Lauterbach and Michael Garland and Shubhabrata Sengupta and David Luebke and Dinesh Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[8] Kirill Garanzha. Efficient Clustered BVH Update Algorithm for Highly-Dynamic Models. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, Los Angeles, USA, 2008.

[9] David W. Gohara. OpenCL Tutorials, Episode 2 - OpenCL Fundamentals. http://www.macresearch.org/opencl_episode2, 2009.

[10] S. Gottschalk, M. C. Lin, and D. Manocha. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. In *SIGGRAPH 96: Conference Proceedings*, pages 171–180, New York, NY, USA, August 1996. ACM.

[11] Scott Le Grand. GPU Gems 3 - Chapter 32. Broad-Phase Collision Detection with CUDA. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch32.html, 2007.

[12] Simon Green. CUDA Particles. NVIDIA CUDA SDK, NVIDIA, November 2007.

[13] Mark Harris, Shubhabrata Sengupta, and John D. Owens. GPU Gems 3 - Chapter 39. Parallel Prefix Sum (Scan) with CUDA. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html, 2007.

[14] Michael Herf. Radix tricks. http://www.stereopsis.com/radix.html, December 2001.

[15] Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-Eui Yoon. HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs. *Computer Graphics Forum (Pacific Graphics)*, pages 1791–1800, 2009.

[16] Peter Kipfer and Rüdiger Westermann. GPU Gems 2 - Chapter 46. Improved GPU Sorting. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html, 2005.

[17] James Klosowski. *Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments*. PhD thesis, State University of New York at Stony Brook, May 1998.

[18] Thomas Larsson and Tomas Akenine-Möller. Strategies for Bounding Volume Hierarchy Updates for Ray Tracing of Deformable Models. MRTC Report, February 2003.

[19] Thomas Larsson and Tomas Akenine-Möller. A Dynamic Bounding Volume Hierarchy for Generalized Collision Detection. *Computers & Graphics*, 2006.

[20] Christian Lauterbach, Sung eui Yoon, and Dinesh Manocha. Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[21] Christian Lauterbach, Qi Mo, and Dinesh Manocha. gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. In *Eurographics*, 2010.

[22] Tomas Möller. A Fast Triangle-Triangle Intersection Test. *Journal of Graphics, GPU, and Game Tools*, 2(2):25–30, 1997.

[23] Aaftab Munshi. The OpenCL Specification. Technical report, Khronos OpenCL Working Group, 2008.

[24] I. J. Palmer and R. L. Grimsdale. Collision Detection for Animation using Sphere-Trees. *Computer Graphics Forum*, 14:105–116, June 1995.

[25] Xavier Provot. Collision and Self-Collision Handling in Cloth Model Dedicated to Design Garments. In *Graphics Interface*, 1997.

[26] Nadathur Satish, Mark Harris, and Michael Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. NVIDIA Technical Report NVR-2008-001, NVIDIA Corporation, September 2008.

[27] Pierre Terdiman. Radix Sort Revisited. http://codercorner.com/RadixSortRevisited.htm, 2000.

[28] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomeranets, and Markus Gross. Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *Proceedings of Vision, Modeling, Visualization VMV'03*, 2003.

[29] Gino van den Bergen. Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics Tools*, 2:1–13, April 1997.