Collision Detection: Broad Phase Adaptation from Multi-Core to Multi-GPU Architecture

Quentin Avril, Valérie Gouranton, Bruno Arnaldi

Collision Detection: Broad Phase Adaptation from Multi-Core to Multi-GPU Architecture

Quentin Avril*, Valérie Gouranton*, Bruno Arnaldi*

*Université Européenne de Bretagne, France
INSA, INRIA, IRISA, UMR 6074, F-35043 RENNES

email: quentin.avril, valerie.gouranton, bruno.arnaldi@irisa.fr

1 Introduction

Collision detection is a well-studied and still active research field in which the main problem is to determine how and if one or more objects collide or will collide in a virtual environment. Many fields are concerned by collision detection, including physical-based simulation, computer animation, robotics, mechanical simulations (medical, biology, cars industry...), haptic applications and video games. In these applications, realtime performance, efficiency and robustness are key issues. In the field of Virtual Reality, physical virtual environments in digital mock-ups and industrial applications are now commonplace, and are of increasingly complexity. The expected level of real time performance is becoming harder to ensure in such largescale virtual environments. Unsurprisingly, collision detection has been an integral part of virtual reality bottlenecks for over thirty years. Recent years have seen impressive advances in collision detection algorithms. However, most algorithms remain unprepared for the new hardware architecture (multi-core, multiprocessor, multi-GPU, etc.). The use of parallel processing has become necessary to take advantage of recent gains of Moores Law. During several years, processors specialists were able to provide clock frequency increases and parallelism improvements in instruction sets. In that way, single threaded applications ran much faster on a new generation of processors without any modification. Now, to have a better management of the power consumption, they promote multi-core architectures. It is no longer possible to rely on the evolution of processing power to overcome the problem of real-time collision detection. The impressive power evolution of graphics hardware and multi-GPU platform is also an important way of algorithm improvements and speed-ups. With these major upheavals in computer architecture it is now essential to take into account run-time architectures to improve collision detection performance. In this paper, we propose new models of collision detection algorithms able to run on new hardware architecture. We focus on three different kind of architecture: multi-core, GPU and multi-GPU.We have developed three new broad phase-based algorithm that take into account the run-time architecture. The rest of our paper is organized as follows: in Section 2 we present the evolution of CPU and GPU these last years. In Section 3 we report related work on collision detection and focus on the multi-core an and GPU-based collision detection algorithms in the parallel programming. Section 4 presents our new multi-core algorithm followed by the Multi-GPU one in Section 5. Both sections show the model and techniques we used to develop the algorithm and also present performances results. We cross results of our new algorithms in Section 6 in order to reveal the limits and differences between them. We then conclude and open on future works in Section 7.

2 Related Work

We present here the collision detection field following by the evolution of CPU and GPU processors. We then present how this evolution has let the setting up of parallel solutions for collision detection to speed-up the computation time.

2.1 Collision Detection

Last decade have seen an impressive evolution of virtual reality applications and more precisely of collision detection algorithms in term of computational

urn:nbn:de:0009-6-348, ISSN 1860-2037

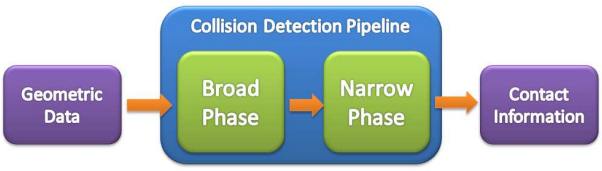

Figure 1: *Collision detection pipeline.*

bottleneck. Collision detection is a wide field dealing with, apparently, an easy problem: determining if two (or several) objects collide. It is used in several domains namely physically-based simulation, computer animation, robotics, mechanical simulations (medical, biology, cars industry), haptics applications and video games. All these applications have different constraints (real-time performance, efficiency and robustness). It has generated a wide range of problems: convex or non-convex objects, 2-Body or N-Body simulations, rigid or deformable objects, continual or discrete methods. Algorithms are also dependent of the geometric model formalism (polygonal, Constructive Solid Geometry (CSG), implicit or parametric functions). All of these problems reveal the diversity of this field of study. For more details we refer to surveys on the topic [LG98, JTT01, TKH+05, KHI+07].

Given n moving objects in a virtual environment, testing all objects pairs tend to perform n^2 pairwise checks. When n is large it becomes a computational bottleneck. Collision detection is represented and built as a pipeline (cf Figure 1) [Hub95]. It is composed by two main parts: broad-phase and narrow-phase. A parallel and adaptive collision detection pipeline running on a multi-core architecture have been proposed [AGA10b]. The goal of this pipeline is to apply successive filters in order to break down the $O(n^2)$ complexity. These filters provide an increasing efficiency and robustness during the pipeline traversal. In the following, we present these parts of the pipeline, broad-phase in section 2.1.1 and narrow-phase in section 2.1.1.

2.1.1 Broad-phase

The first part of the pipeline, called the broad-phase, is in charge of a quick and efficient removal of the objects pairs that are not in collision. Broad-phase algorithms are classified into four main families [KHI+07]:

Brute force approach is based on the comparison of the overall bounding volumes of objects to determine if they are in collision or not. This test is very exhaustive because of its n^2 pairwise checks. A lot of bounding volume have been proposed such as sphere, Axis-Aligned-Bounding-Box (AABB) [Ber97], Oriented-Bounding-Box (OBB) [GLM96] and many others.

Spatial partitioning method is based on the principle that if two objects are situated in distant space sides, they have no chance to collide during the next time step. Several methods have been proposed to divide space into unit cells: regular grid, octree [BT95], quad-tree, Binary Space Partitioning (BSP), k-d tree structure [BF79] or voxels.

Topological methods are based on the positions of objects in relation to others. A couple of objects that are too far one to the other is deleted. *Sweep and prune* is also known as *sort and sweep* [Eri05] being called that way at David Baraff Ph. D thesis in 1992 [Bar92]. Later works like the 1995 paper about I-COLLIDE by Cohen et al. [CLMP95] refer to this algorithm. It is one of the most used methods in the broad-phase algorithms because it provides an efficient and quick pairs removal and it does not depend on the objects complexity. The sequential algorithm of *"Sweep and Prune"* takes in input the overall objects of the environment and feeds in output a collided objects pairs list. The algorithm is divided in two principal parts. The first one is in charge of the bounding volume update of each active virtual objects. Most of time, the bounding volumes used are AABBs that are aligned on the environment axis (cf. Figure 2). The second part is in charge of the detection of overlapping between objects. To do that a projection of higher and upper bounds on the three axis of coordinates of each AABBs is made. Then, we obtain three lists with overlaps pairs on each axis (x, y and z). We can notice two related but different concepts on the way the *Sweep and Prune* operates internally: by starting from scratch each time or by updating internal structures. To differentiate them a name was given to each method, the first type is called brute-force and the second type persistent. A Pair that is still alive after this test mean that its objects are considered as in potential collision. This pair is then transmitted to the narrow-phase.

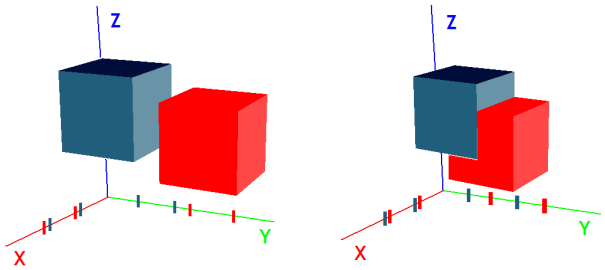

Figure 2: *"Sweep and Prune" algorithm on x and y axis with a non-overlapping condition (left) and an overlapping one (right).*

2.1.2 Narrow-phase

Colliding objects pairs are then given to the narrow-phase that perform an exact collision detection. We can classify narrow-phase algorithms into four main families [KHI+07]:

Feature-based algorithms work on objects primitives: faces (triangle-triangle test [LAM01]), edges and vertices. This family appears in 1991 with the Lin-Canny approach [LC91] or *Voronoï Marching* that proposed to divide space around objects in Voronoï regions that enable to detect closest features pairs between polyhedrons.

Simplex-based algorithms of whom the most famous one is the GJK algorithm [GJK88] that uses Minkowski difference on polyhedrons. Two convex objects collide if and only if their Minkowski difference contains the origin.

Image space-based algorithms work using image-space occlusions queries that are suitable to be used on graphics hardware (GPU). They rasterise objects to perform either 2D or 2.5D overlap test in screen space [BW04]. We further develop this part in the parallel section.

Bounding volume-based algorithms are used in most of strategies to perform collision tests and it highly improves performances. Bounding volume hierarchies (BVH) allow arranging bounding volume into a tree hierarchy (binary tree, quad tree...) in order to reduce the number of tests to perform. A description on these BVH and a comparison between their performance can be found in [Eri05]. Deformable objects are very challenging for BVH because hierarchy structures have to be updated when an object deforms itself [Ber97, TKH+05].

3 Architecture Evolution

We briefly present in this section, the evolution of CPU and GPU these last years. We first describe the emergence and spread of multi-core processors, followed in a second step by the impressive evolution of GPU in term of computation power and ease of use.

3.1 From Sequential CPU to Multi-core Architecture

Compared to actual outlook, it seems clear that Gordon Moore was a lucky man. Since 1965, he predicts a duplication of the number of transistors on a microprocessor each two years. During more than forty years, this guesswork seems exact but we know now that physical limits (power and heat) prevent this duplication. What is the solution to keep alive Moore law? You make more cores. Nowadays trend tends to be duplication of cores in computers and parallel architecture. The first personal computer with a core-duo arrived in 2005 with AMD[1] followed by Intel[2]. In 2006 Sun presented its new octo-core called Niagara2. Intel presents last year a 32 in order x86 cores [SCS+08] called "Larrabee" and Sun recently presents 80 cores computer and it seems that new trends are not only at the multi-core but also at the many-core. Difference between these types of cores is the start and stop notion on the way, if you need n cores to work, computer will only starts n cores. Many-core is very useful because when people need not the entire power of cores, computer turns off some of them. Until now, 3D objects and virtual environments grew up parallel to processor power, so researchers were continuously looking for an improvement of the collision detection algorithms in order to increase their precision and robustness. A lot of articles still continue to improve collision detection algorithms these last recent years. But now, processors power stay the same while virtual environments are more and more sized, so new trends are not only in the algorithms improvement but also in the algorithms architecture modification. As we can not hope a continual evolution of processors we have now to study how it is possible to use multi-core in collision detection algorithms. Nowadays it is impossible to present CPU without dealing with central memory handling; on a multi or many cores there is a very complex cache handling between cores and this handling

[1] www.amd.com
[2] www.intel.com

is continually improved to increase computer performance. Cache and memory handling is another point that cannot be ignored in the optimization of the collision detection performance.

3.2 From Graphic Processor to GPGPU

Recent years have seen the evolution of graphics hardware from fixed function units toward an increasingly programmable graphics pipeline. Contrary to CPU, Graphics Processing Unit (GPU) has a very important power evolution since few years. This impressive evolution can be explain by the way that in one hand, CPU is a generalist processor which deals with ordinary data which are often dependent, several of its components are in charge of the data stream control and its memory latency period is hidden by data caching. In the other hand GPU provides processor well-suited to highly parallelizable computations, it deals with independent data so it needs not a sophisticated data stream control and its memory latency period is hidden by computations. General-purpose Processing on Graphics Processing Unit is the technique allowing graphics hardware (GPU) to perform computations traditionally reserved to CPU. A survey has been published [OLG+07] on GPGPU focusing on a simple presentation of GPGPU applications. Using graphics cards in order to increase mathematical computations is not recent. During the nineties, some researchers use rasterizer and Z-Buffer of the graphics cards to accelerate path, for instance, path finding or Vorono printing. But revolution appears in 2003 with evolved shaders allowing matrix computations on graphics cards. From this year, a SIGGRAPH section is dedicated to this new computation technique. To handle GPU in 2003, OpenGL or Direct3D were essential. Brook was the first C language extension that allowed using GPU as a co-processor for parallel computations. In 2007, Nvidia3 developed a language and a software called CUDA (Compute Unified Device Architecture) exploiting GPUs power, using principles of parallel programming with threads. This API can be seen as a C language extension and its assembly language is PTX. ATI/AMD develops its own language for graphics cards, called Brook+. Runtime uses CAL for the GPU backend. Even if AMD technology is as efficient as Nvidias (or even more), Brook+ is less used than CUDA, due to a lack of documentation on it and to a higher difficulty to code solution.

3.3 Parallel Collision Detection

The parallel solution of collision detection algorithms is a recent field in high performance computing. We can distinguish three different families of algorithms, namely: GPU-based, CPU-based and hybrid-based.

3.3.1 GPU-based algorithms

The GPU-based family is used to perform collision detection for few years using typical GPU solutions but it becomes more and more used to perform non-common GPU solutions. We call "typical GPU solutions", the algorithms that are based on the image-space. Image space-based algorithms work using image-space occlusions queries that are suitable to be used on graphics hardware. They rasterize objects to perform either 2D or 2.5D overlap test in screen space [BW04]. Non-common GPU solutions are more recent ones generally developed with CUDA and not using image space to detect collisions.

Cinder [KP03] is an algorithm exploiting GPU to implement a ray-casting method to detect static interference between solid polyhedral objects. The algorithm is linear in relation to the number of objects and number of polygons in the environment. It also requires no preprocessing or special data structures. Other methods have been proposed using ray-casting, Hermann et al. [HFR08] use it to detect collision and to create contact forces. GPU-based algorithms for self-collision and cloth animation have also been introduced by Govindaraju et al. [GLM05a, GLM05b]. Juarez-Comboni et al. [JMJC05] describe the use of several GPUs during collision detection process. One GPU is in charge of the collision detection process using a simple boundary volume collision query. The other one is in charge of the rendering operations. An algorithm using *Layered Depth Images* (LDI) to detect collision and create physical reaction, has been proposed [FBAF08]. It has been developed to run on a single GPU. An LDI is a representation and rendering method for objects. Similar to a two-dimensional image, the LDI consists of an array of pixels. Contrary to a 2D image, an LDI pixel has depth information and there are multiple layers at a pixel location. LDI algorithm has been introduced by Shade & al [SGwHS98] to represent multiple geometric layers from one viewpoint. Heidelberger et al. [HTG03, HTG04] have extended the model of LDI to build geometrical models of volume intersections. A solution using image-space visibility queries has been proposed for the broad

phase [GRLM03].

A recent work uses thread and data parallelism on a single GPU to perform fast hierarchy construction, updating, and traversal using tight-fitting bounding volumes such as oriented bounding boxes (OBB) and rectangular swept spheres (RSS) [LMM10]. We have also proposed a solution based on a GPU mapping function that enables GPU threads to determine the objects pair to compute without any global memory access using a square root approximation technique based on Newtons estimation [AGA12].

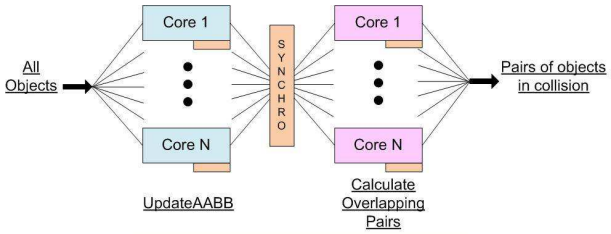

Figure 3: *Our parallel broad-phase algorithm. Parallelization of the update AABB part and the calculate overlapping pair one with a synchronization point between them.*

3.3.2 CPU-based algorithms

The pipeline has never been parallelized but Zachmann [Zac01] made an evaluation of the performance of a theoretical parallelized back-end of the pipeline and showed that if the environment density is large compared to the number of processors, then good speed-ups can be noticed. Multi-processor machines are also used to perform collision detection [KS95]. Depth-first traversal of bounding volumes tree traversal (BVTT) and parallel cloth simulation [SSIF09] are good instances of this kind of work. Dodier et al. [DLAG13] have proposed a distributed and anticipative model for collision detection on distributed systems such as PC clusters. Their model allows to break synchronism constraints for the collision detection process that allows the simulation to run in a decentralized and distributed fashion.

Few papers appeared dealing with new parallel collision detection algorithms using multi-core architecture. A new task splitting approach for implicit time integration and collision handling on a multi-core architecture has been proposed [TPB08]. Tang et al. [TMT08] propose to use a hierarchical representation to accelerate collision detection queries and an incremental algorithm exploiting temporal coherence. The overall is distributed among multiple cores. They obtained a 4X-6X speed-up on a 8-core processor based on several deformable models. Kim et al [KHeY08] propose to use a feature-based bounding volume hierarchy (BVH) to improve performances of continuous collision detection. They also propose novel task decomposition methods for their BVH-based collision detection and dynamic task assignment methods. They obtained a 7X-8X speed-up using a 8-core architecture compared to a single-core. Hermann et al. [HRF09] propose a parallelization of interactive physical simulations. They obtain a 14X-16X speed-up on a 16-core architecture compared to a single-core.

3.3.3 Hybrid-based algorithms

More and more papers appear dealing with new hybrid solutions that run on multi-core and multi-GPU architecture. Kim et al. [KHH+09] have presented an hybrid parallel continuous collision detection method *HPCCD* based on a bounding volume hierarchy. Recently, Pabst et al. [PKS10] have presented a new hybrid CPU/GPU method for rigid and deformable objects based on spatial subdivision. Broad and narrow phases are both executed on a multi-GPU architecture.

3.4 Positioning

Related work lets appear that many studies have been made to improve efficiency and performance of collision detection algorithms. The use of parallelism is becoming commonplace to address the problem of real-time collision detection [AGA09]. Thus, only fine-grain parallelizations have been done on algorithms and, for the moment, there is no work on a global parallelization of the pipeline stages and on its adaptation on any number of cores.

4 Multi-Core Broad Phase

The architecture of collision detection algorithms needs to be improved to face real-time interaction. In this way, we focus on an essential step of the collision detection pipeline: the broad-phase. More precisely, our algorithm is an implementation of the *"Sweep and Prune"*[CLMP95] on a multi-core architecture [AGA10a].

4.1 Multi-Threaded Algorithm

Multi-core architecture enable to separate collision detection computations on available cores. But computations can not be separated on the way without a special data structure. To fully exploit multi-core architecture, critical sections, threads idling and cores synchronization have to be taken account and minimized or avoided. To parallelize the algorithm we have decided to use OpenMP[3] because of the directives that allow to keep the same code (with few algorithmic modifications on the data structure) and to focus on the directives. Even if IntelTBB provides better performances, it is more complex to program with and it generates specific code, enable to work without IntelTBB libraries.

A simplified scheme of our model is in Figure 3. We can notice parallelization of the two principal parts of the algorithm with a synchronization between both. Number of threads that are created depends on the number of available cores. As a thread is only in charge of geometric computations and does not wait for anything, create more than one thread per core will increase computation time. In the first step of the algorithm, each thread works on $\frac{n}{c}$ objects where n is the number of objects in the environment and c the number of cores. It is possible to divide objects per threads because AABB update computation does not depend of the object complexity, time spent per object by a thread is almost homogeneous. Compared to the sequential algorithm where new computed bounding volume is written on the way in a data structure, we can not use the same scheme without avoiding critical writing section between threads. That is why we introduce a new smallest data storage used by each thread to put new computed bounding volume. This new structure is an array dynamically allocated in relation to the number of cores and objects. Synchronization between this two steps is compulsory to merge all the new bounding volumes in the same data structure. We only merge threads array pointers to reduce synchronization time.

In the second part of the algorithm, each thread works on $\frac{(n^2-n)}{2}/c$ pairs of objects where c is still the number of cores. Like in the first part, each computation made by a thread is an overlapping test between objects coordinates so it does not depend on the object complexity. To avoid critical section between threads we use a similar technique where each thread is fitted with its own data storage to put objects pairs with overlapped coordinates. All pairs of objects in collision are merged at the end of the overall computation to create the pair list of objects in collision. Then, this new pairs list is given to the narrow-phase that performs an exact collision detection test. This kind of broad phase algorithm is well-suited to the parallelization because there is no dependency between computations. They can be distributed among 2, 4, 8 or more cores without disturbing results.

[3]OpenMP - http://openmp.org/wp/

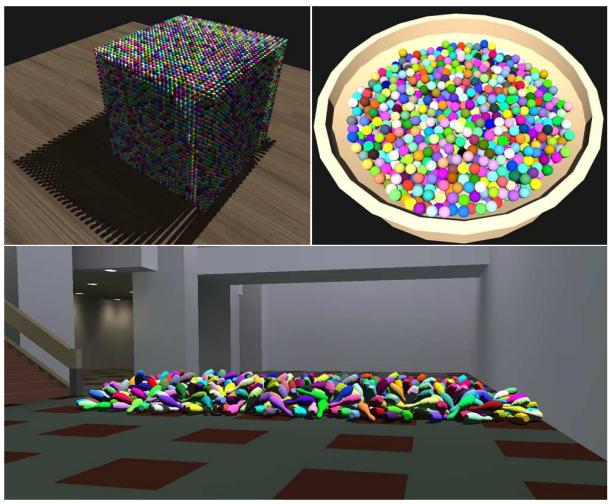

Figure 4: ***Benchmarks:*** *We used several benchmark models to measure collision detection time: 10K balls of 2K polygons each falling in simple environment of 600 polygons (= 1.1M polygons), 20K cubes of 12 polygons each fallen on complex environment of 300K (= 420K polygons) and 3.5K concave shapes (skittles of 20K each) falling on a plan. We only performed test on n-body simulation of rigid bodies using AABB as bounding volume.*

4.2 Results

In this section we present main results of computation time speed-up. Those tests were performed through several benchmark models (cf Figure. 4). We only performed test on *n*-body simulation of rigid bodies using AABB as bounding volume. To obtain homogeneous results, we have only worked on a 8-cores computer using 1, 2, 4 or 8 cores. We work on Windows XP Professional x64 Edition Version 2003 with Intel Xeon (2*Quad) CPU X5482 of 3.20 GHz and with 64 GB of RAM.

We present here time results for all used benchmark models (Cubes, Balls and Skittles). Numerical results

	Cubes	Balls	Skittles
1 core	8,89ms	4,45ms	1,6ms
2 cores	4,96ms	2,48ms	0,9ms
4 cores	2,76ms	1,4ms	0,5ms
8 cores	1,52ms	0,74ms	0,27ms

Figure 5: *Time spent for updating AABB for each benchmark model from 1 core to 8 cores.*

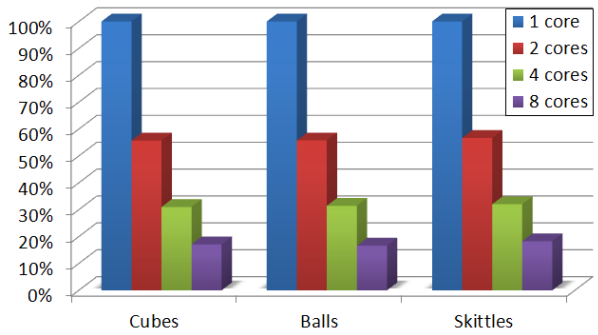

Figure 6: *The AABB update execution time in relation to the number of cores. The overall computation time is reduced by 17.03% by using 8 cores on this benchmark.*

	Cubes	Balls	Skittles
1 core	53,339ms	26,7ms	10,71ms
2 cores	31,65ms	15,748ms	6,35ms
4 cores	18,76ms	9,51ms	3,742ms
8 cores	11,43ms	5,82ms	2,314ms

Figure 7: *Time spent to calculate overlapping pairs for each benchmark model from 1 core to 8 cores.*

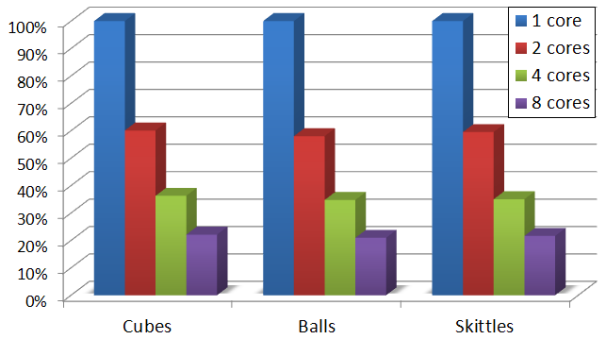

Figure 8: *The execution time of the overlapping pairs checks in relation to the number of cores. The overall computation time is reduced by 21.56% by using 8 cores on this benchmark.*

for the first part of the algorithm is presented in tab 5. The reduction of the overall running time is shown on the graphic in Figure 6. We can see a percentage of time reduction for the first part of the algorithm concerning the AABB update. For one scenario four blocks show time spent from 1 to 8 cores and we can notice that time decreases when the number of cores goes up. The overall running time is reduced by 56.04% by using 2 cores, 31.49% for 4 cores and 17,03% for 8-cores. Numerical results for the second part of the algorithm is presented in tab 7. This second part of the algorithm is shown in the graphic Figure 8 and we notice the same gain of time than the first part. The overall running time is reduced by 59.2% by using 2 cores, 35.34% for 4 cores and 21.56% for 8-cores.

The general speed-up of our parallel algorithm is shown in Figure 9, on this graphics our work is represented by the pink line bounded by the blue one which is the optimal speed-up for a parallel execution whose we wanted to get closer. We have also performed measures on the computation time spent by t threads shared on c cores and the assumption made at the beginning on using more than one thread per core seems to be exact. Time spent by 3 threads on 2 cores is slower than 2 threads but better than 1. So using more than one thread per core is not justified and appears to be less efficient.

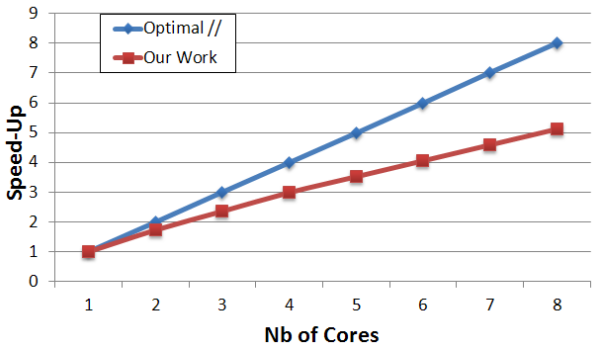

Figure 9: *The overall gain of the execution. A speed-up of 5,1 is obtained on a 8-cores computer.*

4.3 Positioning Key

We have presented a new way to parallelize the *"Sweep and Prune"* algorithm on a multi-core architecture. Results show that our solution enables to reduce computation time by almost 5X-6X on a 8-cores architecture. The persistent method that updates an internal structure is still more interesting compared to the brute force one parallelized on 2 or 4 cores but be-

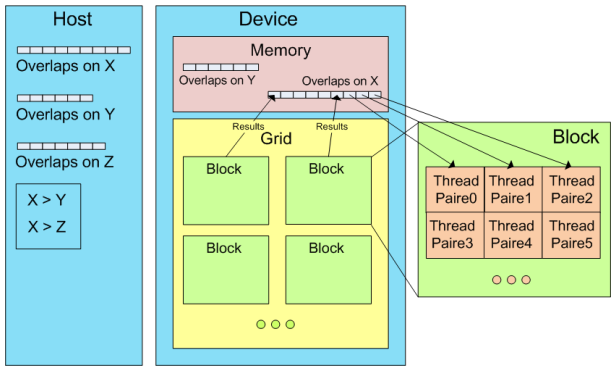

Figure 10: *"Sweep and Prune" algorithm on a single GPU. Each pair of the biggest tab is handled by a thread that looks for similar pair in the other input tab.*

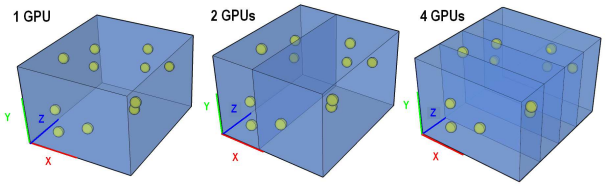

Figure 11: *Example of spatial subdivision used for multi-GPU "Sweep and Prune" algorithm. We seek the axis with the largest number of overlapping pairs and subdivide this axis. We then create a CPU thread by area in charge of one GPU device to perform the algorithm in its area.*

comes longer compared to the 8-cores parallelization. As processors will soon have more and more cores, using the brute force broad phase algorithm will become a necessity to take full advantage of these highly parallelizable architecture. GPU is also subjected to an impressive evolution of its number of cores.

5 Multi-GPU Broad Phase

We continue by presenting a new way to parallelize the broad phase algorithm on a multi-GPU architecture. First, we describe the existing algorithm we used and then our new model running on a multi-core and multi-GPU architecture.

5.1 GPU "Sweep and Prune"

We have started the development with a first implementation of this broad phase algorithm on a single GPU. The algorithm is divided in three parts which two of them are executed by the GPU. The first part is in charge of determining which pairs of object are in overlapping. On the CPU we maintain three sorted lists of starts (lower bound) and ends (upper bound) of objects bounding volume which we extract overlapping pairs. GPU is in charge of extracting pairs common to all three lists (cf Figure 10). This work is done by a CUDA algorithm that assigns to each GPU threads a kernel function in charge of extracting pairs in a smaller dataset. We first compare X and Y axis creating a tab results in the GPU memory that corresponds to pairs that are in both input axis. To optimize performances we check before separating data between threads which axis is the "fullest" one, in other ways which tab is the biggest one. A thread is created for each pair of this axis, and each thread is in charge of determining if there is a similar pair in the other input axis. Then we compare the Z axis with the previous tab results.

5.2 Spatial Subdivision for Multi-GPU

After adapting the *"Sweep and Prune"* algorithm on a GPU architecture, we now present how it is possible to adapt it on a multi-GPU architecture. Difference between these two versions is in the genericity of the second one because it is able to work on a n-GPU platform. To separate computations between GPU devices during the broad phase process we use dynamic spatial subdivision and more precisely we divide space by the number of GPUs. The subdivision technique is not a regular one as are grids or octrees but depends on the density distribution of objects in the environment. As the computational complexity of the algorithm only depends on the number of objects in the scene, we can decompose the environment from the density of objects. This repartition enables to balance GPU's computation time and obtain an homogeneous one between GPUs. Figure 11 presents the technique we used to subdivide environment and distribute computations between GPU devices. We check among axis which one has more overlapping pairs, then we divide it by the number of GPUs in order to separate homogeneously number of overlapping pairs between them. Each GPU is now in charge of looking for overlapping pairs in its own data set. As we mentioned in the overview each GPU is managed by a CPU core to provide a global parallelization on multi-GPU and multi-core. This is done by using OpenMP, which is a

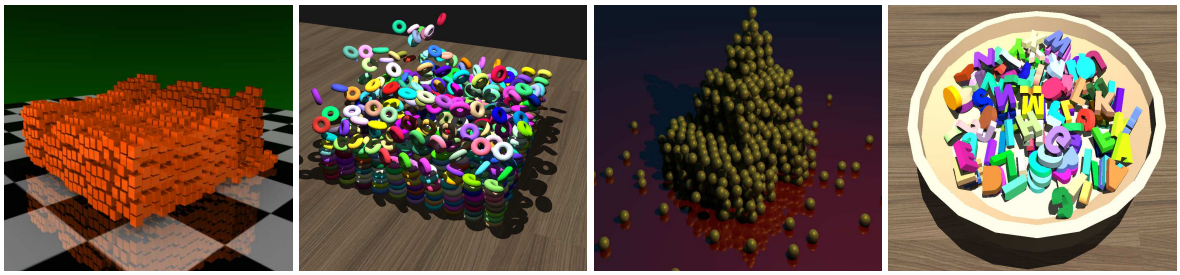

Figure 12: **Benchmark**: Four virtual environments used during simulation tests - (a) Cubes - (b) Torus - (c) Spheres - (d) Alphabet letters.

Models	Nb Objects	Nb Polygons	Properties
Cubes	20.000	240.000	Convex and Simple
Balls	10.000	10.800.000	Convex and Complex
Torus	1.000	2.304.000	Concave and Similar
Alphabet	260	31.680	All Differents

Figure 13: *Geometric and numerical properties of our four benchmark environments.*

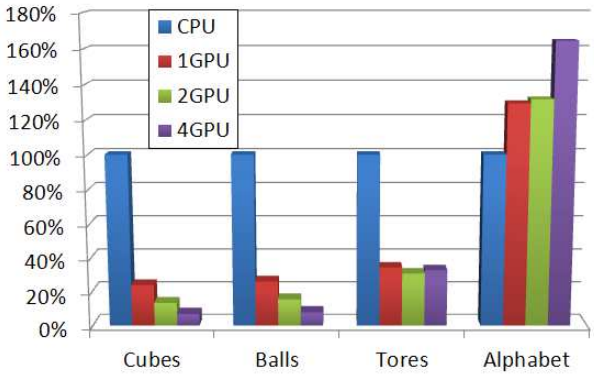

Figure 14: *The execution time (compared in % to the CPU time) of the broad phase process in relation to the run-time architecture.*

parallelization standard allowing to parallelize the execution on several cores by using compiler directives. Each thread on a core is in charge of a part of the global environment and of its GPU that executes the broad phase algorithm.

At the end we synchronise every GPU results to create the list of object pairs to transmit to the narrow phase.

5.3 Results

We tested our new collision detection pipeline with different simulation scenarios, going from similar objects that are completely independent to heterogeneous scenes of colliding objects (cubes, balls, torus and alphabet letters) (cf Figure 12 and 13). Tests were performed on a 4 * Quadro FX 4600 with Intel(R) Xeon(R) CPU X5482 @ 3.20 Ghz (Octo-core) on Windows XP(v64) with 64GB of RAM.

Graphic 14 presents computation time during the broad phase process on our four benchmark tests. We measured time spent by four algorithms (from sequential CPU to four GPUs). We can notice a significant difference between CPU and GPU and also between using 1, 2 or 4 GPUs. For large-scale virtual environment speed-up is very significant whereas results show that using 4 GPUs to perform a small scale environment brings a loss of time. For example with the first benchmark (20.000 Cubes) using one GPU reduces time by 4,2 in relation to the CPU computation time. Time spent by the algorithm on CPU is here to compare with GPU measures but it is a non performant time because of the brute force method. Using this CPU algorithm during the broad phase process if you only have a sequential CPU is highly not recommended. We use it because this is the most parallelizable broad phase algorithm. The use of 2 GPUs reduces time by 1,79 in relation to the use of one single GPU and 4 GPUs reduces it by more than 3,5.

On the contrary in the last benchmark (Alphabet), CPU time is the best one because there is only few objects and the broad phase algorithm is linear with number of object and does not take into account object complexity. Results show that using one GPU allow to significantly reduce computation time during the broad phase process into large scale environment. Results also show that multi-GPU solution is perfectly suited for this kind of highly parallelizable algorithm and allow to divide computation time on 2 and 4 GPUs architecture. Results has also shown that using the

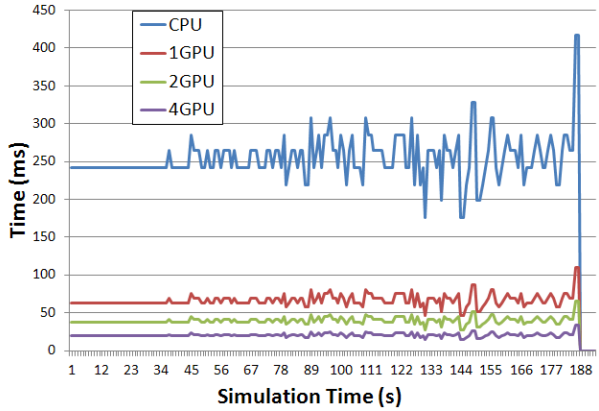

Figure 15: *Test made with the "balls" environment to compare algorithms behaviors throughout the simulation. Tests were performed from sequential CPU to 4 GPUs during the broad phase process.*

largest number of available GPU might not insure you the best performances when using small scale environment.

Graphic 15 shows performance measurements of the broad phase process during the "balls" simulation. We did four time the same simulation but with four different algorithms from sequential CPU to 4 GPUs. We can see on this graphic that algorithms has the same computations time changes all along the simulation, these changes are related to the simulation evolution. The horizontal line at the beginning of each curve represents the fall of balls before dropping on to the floor.

6 Conclusion

We have presented several contributions on the collision detection optimization centered on hardware performance. We focus on the first step (Broad-phase) and propose three new ways of parallelization of the well-known *Sweep and Prune* algorithm. We first developed a multi-core model takes into account the number of available cores. Multi-core architecture enables us to distribute geometric computations with use of multi-threading. Critical writing section and threads idling have been minimized by introducing new data structures for each thread. Programming with directives, like OpenMP, appears to be a good compromise for code portability. We then proposed a new GPU-based algorithm also based on the *"Sweep and Prune"* that has been adapted to multi-GPU architectures. Our technique is based on a spatial subdivision method used to distribute computations among GPUs. Results show that significant speed-up can be obtained by passing from 1 to 4 GPUs in a large-scale environment.

Results suggest a multitude of future directions. It could be interesting to focus on repartition techniques that can be used to distribute data and tasks between GPUs to determine which one is the most suitable for a multi-GPU platform. Specifically, there is still room for improvement in the field of data division during the exact collision detection step (narrow phase). The *Sweep and Prune* algorithm can also be parallelized in many ways by proceeding to a different division of axis. We saw that using 4 GPUs in a small scale environment brings a loss of time. Another way of optimization could be an evaluation of the most suitable number of GPU to use to obtain best performances, as using all available GPUs during physical simulations might not insure best performance. Multi-GPU technique is going to be a key component of parallel collision detection algorithm. The design of such systems requires a detailed analysis of task and data repartition techniques to optimize the performance of these complex runtime architectures.

7 Acknowledgements

This work would not have been possible without the help of several people who provided great help and our beautiful region of Brittany who provided funding (ARED financing - $GriRV$ Project N°4295). This paper is related to a *Best Student Paper Award* received on April 2010 at the VRIC conference, the authors thank the conference's organisers and people who voted for our work.

References

[AGA09] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi, *New trends in collision detection performance*, Virtual Reality International Conference (VRIC) 2009 (Simon Richir & Akihiko Shirai, ed.), April 2009, pp. 53–62.

[AGA10a] _____, *A broad phase collision detection algorithm adapted to multi-cores architectures*, Virtual Reality International Conference (VRIC) 2010 (Simon Richir

& Akihiko Shirai, ed.), April 2010, pp. 95–100.

[AGA10b] ———, *Synchronization-free parallel collision detection pipeline*, International Conference on Artificial Telexistence (ICAT) 2010, December 2010.

[AGA12] ———, *Fast collision culling in large-scale environments using gpu mapping function*, Eurographics Symposium on Parallel Graphics and Visualization (2012) (Cagliari, Italy) (Hank Childs, Torsten Kuhlen, and Fabio Marton, eds.), Eurographics Association, 2012, pp. 71–80.

[Bar92] David Baraff, *Dynamic simulation of non-penetrating rigid bodies*, Ph.D. thesis, Cornell University, 1992.

[Ber97] Gino Van Den Bergen, *Efficient collision detection of complex deformable models using aabb trees*, J. Graph. Tools **2** (1997), no. 4, 1–13.

[BF79] Jon Louis Bentley and Jerome H. Friedman, *Data structures for range searching*, ACMCS **11** (1979), no. 4, 397–409.

[BT95] Srikanth Bandi and Daniel Thalmann, *An adaptive spatial subdivision of the object space for fast collision detection of animated rigid bodies*, Comput. Graph. Forum **14** (1995), no. 3, 259–270.

[BW04] George Baciu and Wingo Sai-Keung Wong, *Image-based collision detection for deformable cloth models*, IEEE Trans. Vis. Comput. Graph **10** (2004), no. 6, 649–663.

[CLMP95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi, *I-collide: An interactive and exact collision detection system for large-scale environments*, SI3D, 1995, pp. 189–196, 218.

[DLAG13] Steve Dodier-Lazaro, Quentin Avril, and Valérie Gouranton, *SODA: A scalability-oriented distributed & anticipative model for collision detection in physically-based simulations*, GRAPP/IVAPP (Sabine Coquillart, Carlos Andújar, Robert S. Laramee, Andreas Kerren, and José Braz, eds.), SciTePress, 2013, pp. 337–346.

[Eri05] Christer Ericson, *Real-time collision detection*, Morgan Kaufmann, 2005.

[FBAF08] François Faure, Sébastien Barbier, Jérémie Allard, and Florent Falipou, *Image-based collision detection and response between arbitrary volumetric objects*, September 12 2008.

[GJK88] Elmer G. Gilbert, Daniel W. Johnson, and Sathiya S. Keerthi, *A fast procedure for computing the distance between complex objects in three-dimensional space*, IEEE Journal of Robotics and Automation **4** (1988), 193–203.

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha, *Obbtree: A hierarchical structure for rapid interference detection*, Proceedings of the ACM Conference on Computer Graphics (New York), ACM, August 4–9 1996, pp. 171–180.

[GLM05a] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha, *Fast and reliable collision detection using graphics processors*, COMPGEOM: Annual ACM Symposium on Computational Geometry, 2005.

[GLM05b] ———, *Quick-cullide: fast inter- and intra-object collision culling using graphics hardware*, SIGGRAPH '05: ACM SIGGRAPH 2005 Courses (New York, NY, USA), ACM, 2005, p. 218.

[GRLM03] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha, *Cullide: Interactive collision detection between complex models in large environments using graphics hardware*, SIGGRAPH/Eurographics Workshop on Graphics Hardware (San Diego, California) (M. Doggett, W. Heidrich, W. Mark, and A. Schilling, eds.), Eurographics Association, 2003, pp. 025–032.

[HFR08] Everton Hermann, Francois Faure, and Bruno Raffin, *Ray-traced collision detection for deformable bodies*, GRAPP, 2008, pp. 293–299.

[HRF09] Everton Hermann, Bruno Raffin, and François Faure, *Interactive physical simulation on multicore architectures*, Eurographics Workshop on Parallel and Graphics and Visualization, EGPGV'09, March, 2009 (Munich, Allemagne), 2009.

[HTG03] Bruno Heidelberger, Matthias Teschner, and Markus H. Gross, *Real-time volumetric intersections of deforming objects*, VMV (Thomas Ertl, ed.), Aka GmbH, 2003, pp. 461–468.

[HTG04] _____, *Detection of collisions and self-collisions using image-space techniques*, WSCG, 2004, pp. 145–152.

[Hub95] Philip M. Hubbard, *Collision detection for interactive graphics applications*, IEEE Transactions on Visualization and Computer Graphics **1** (1995), no. 3, 218–230, ISSN 1077-2626.

[JMJC05] Andy M. Day Jose M. Juarez-Comboni, *A multi-pass multi-stage multi-gpu collision detection algorithm*, Graphicon 2005 Proceedings, 2005.

[JTT01] Pablo Jiménez, Federico Thomas, and Carme Torras, *3d collision detection: a survey*, Computers & Graphics **25** (2001), no. 2, 269–285.

[KHeY08] DukSu Kim, Jea-Pil Heo, and Sung eui Yoon, *Pccd: Parallel continuous collision detection*, Tech. report, Dept. of CS, KAIST, 2008.

[KHH+09] Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-Eui Yoon, *HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs*, Comput. Graph. Forum **28** (2009), no. 7, 1791–1800.

[KHI+07] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and Richard Rowe, *Collision detection: A survey*, Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on (2007), 4046–4051.

[KP03] Dave Knott and Dinesh K. Pai, *Cinder: Collision and interference detection in real-time using graphics hardware*, Graphics Interface, 2003, pp. 73–80.

[KS95] Yoshifumi Kitamura and Andrew Smith, *Parallel algorithms for real-time colliding face detection*, Robot and Human Communication (1995), 211–218 (en).

[LAM01] Thomas Larsson and Tomas Akenine-Mller, *Collision detection for continuously deforming bodies*, Eurographics (2001) (en).

[LC91] Ming C. Lin and John F. Canny, *A fast algorithm for incremental distance calculation*, Tech. report, University of Berkeley, California, March 19 1991.

[LG98] Ming C. Lin and Stefan Gottschalk, *Collision detection between geometric models: a survey*, Proceedings of the 8th IMA Conference on the Mathematics of Surfaces (IMA-98) (Winchester, UK) (Robert Cripps, ed.), Mathematics of Surfaces, vol. VIII, Information Geometers, September 1998, pp. 37–56.

[LMM10] C. Lauterbach, Q. Mo, and D. Manocha, *gproximity: Hierarchical gpu-based operations for collision and distance queries*, Computer Graphics Forum (EUROGRAPHICS Proceedings), vol. 29, June 2010, pp. 419–428.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell, *A survey of general-purpose computation on graphics hardware*, Proceedings of Eurographics 2004, Blackwell Publishing Ltd, 2007, Warning: the year was guessed out of the URL., pp. 21–51 (en).

[PKS10] Simon Pabst, Artur Koch, and Wolfgang Straβer, *Fast and scalable cpu/gpu*

collision detection for rigid and deformable surfaces, Computer Graphics Forum, vol. 29, July 2010, pp. 1605–16212.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan, *Larrabee: a many-core x86 architecture for visual computing*, ACM SIGGRAPH'08 Transactions on Graphics **27** (2008), no. 3.

[SGwHS98] Jonathan Shade, Steven J. Gortler, Li wei He, and Richard Szeliski, *Layered depth images*, SIGGRAPH, 1998, pp. 231–242.

[SSIF09] Andrew Selle, Jonathan Su, Geoffrey Irving, and Ronald Fedkiw, *Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction*, IEEE Trans. Vis. Comput. Graph **15** (2009), no. 2, 339–350.

[TKH+05] Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrmann, Marie-Paule Cani, François Faure, Nadia Magnenat-Thalmann, Wolfgang Straßer, and Pascal Volino, *Collision detection for deformable objects*, Comput. Graph. Forum **24** (2005), no. 1, 61–81.

[TMT08] Min Tang, Dinesh Manocha, and Ruofeng Tong, *Multi-core collision detection between deformable models*, Computers & Graphics, 2008.

[TPB08] Bernhard Thomaszewski, Simon Pabst, and Wolfgang Blochinger, *Parallel techniques for physically based simulation on multi-core processor architectures*, Computers & Graphics **32** (2008), no. 1, 25–40.

[Zac01] Gabriel Zachmann, *Optimizing the collision detection pipeline*, Proc. of the First International Game Technology Conference (GTEC), January 2001.