Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2010)
J. Bender, K. Erleben, and M. Teschner (Editors)

Accelerated Hierarchical Collision Detection for Simulation using CUDA

Jimmy A. Jorgensen., Andreas R. Fugl and Henrik G. Petersen

Maersk Mc-Kinney Moller Institute at University of Southern Denmark, Denmark

Abstract

In this article we present a GPU accelerated, hybrid, narrow phase collision detection algorithm for simulation purposes. The algorithm is based on hierarchical bounding volume tree structures of oriented bounding boxes (OBB) that in the past has shown to be efficient for collision detection.

The hierarchical nature of the bounding volume structure complicates an efficient implementation on massively parallel architectures such as modern graphics cards and we therefore propose a hybrid method where only box and triangle overlap tests and transformations are offloaded to the graphics card.

When exploiting coarse-grained parallelism in grasping and stacking simulations, requiring all-contacts resolution, a performance gain of up to 7x compared to the collision detection package PQP is obtained.

1. Introduction

Collision detection between geometric models is known to be a fundamental task in many areas such as entertainment, robot motion planning and physics simulation. Applications include haptic rendering, 3D games, animations, motion planning and more.

For most of these applications collision detection is often the performance bottleneck. This has motivated extensive research in efficient collision detection algorithms. Lately the scientific community has shown an increased interest for accelerating algorithms on specialized hardware with parallel computing capabilities. Part of the explanation of this new focus is the broad range of new computing platforms that are cheap, available and that promise high theoretical performance.

One such platform is the modern graphics processing unit (GPU). Originally used in the demanding 3D gaming industry, the GPU has become popular for general purpose computing, resulting in a community around the technology (http://www.gpgpu.org).

In this paper we present a collision detection algorithm for the GPU. It is primarily designed for but not limited to applications of rigid body simulation. Because of the massively parallel architecture of the GPU a hybrid method has been developed, where administrative tasks are executed on the CPU while the GPU performs administratively simpler and computationally heavier tasks.

The method is tested against the Proximity Query Package (PQP) [GLM96] which is an optimized and available software library for collision detection. We test an application scenario in both serial and parallel execution of collision queries, where all contacts between objects needs to be resolved.

Our method show a performance gain of up to a factor of 7 in applications where collision queries are run in parallel.

2. Related work

Considerable work on developing efficient and specialized algorithms for collision detection has been done in the last two decades. Being generally flexible and efficient, algorithms based on hierarchical bounding volume structures has received special focus [GLM96, KHM*98, CLMP95] to mention a few.

Within this special class of collision detection algorithms, the following areas has been subject to extensive optimisation: overlap tests of primitives [Hel97, DG02] and bounding volumes [GLM96], temporal coherence when using separating axis theorem (SAT), temporal coherence on the bounding volume test tree [TTSD06], spatial coherence, hybrids on the hierarchical data structure, cache friendly data struc-

DOI: 10.2312/PE/vriphys/vriphys10/097-104

tures [YM06], Non-binary tree structures and lately usage of parallel architectures.

In [NA05] they present a naive implementation of collision detection in an FPGA, based purely on triangle intersection tests which is later combined with a dedicated motion planning chip [NA06]. Recently complete hierarchical collision detection kernels has been implemented in dedicated hardware. In [RHZA06a] and [RHZA06b] hierarchical collision detection based on k-dops with fixed-point arithmetic was implemented and later optimized for memory bandwidth. Unfortunately only vague comparison to an unknown algorithm on the PC platform was presented.

The earliest collision detection algorithms on GPUs used the specialised 3D rendering pipeline to utilise image-space computations which were applicable to both rigid and deformable models [BWS98, BW03, HTG04]. With advances and more openness of the GPU architecture it became easier to write general purpose algorithms [OLG*07]. First came the introduction of specialised programs known as "shaders" and later came full access to the parallel "many-core" architecture of the modern GPU. Though languages for programming the GPUs are still vendor specific, with CUDA from NVIDIA and CTM from ATI in the lead, steps are taken toward vendor independent languages such as OpenCL and BrookGPU.

Latest research in collision detection algorithms for modern GPU's have focused on broad phase implementations, particle simulation and deformable models [Ngu07] which all fit relatively well to the many-core architecture.

However recently a hybrid approach similar to that described in this paper was presented in [KHH*09]. Their method is based on a continuous collision strategy and they obtain a speedup similar to that presented in this paper. This however is with the use of 2 GPU and 4 CPU cores in comparison to our method that use 1 GPU and 1 CPU core.

Similar but more comprehensive work has later been done in [LMM10] where both discrete, continuous and distance queries where implemented on the GPU. Instead of the hybrid approach the complete algorithm was implemented on the GPU which for discrete collision detection provided similar results as those obtained in this work.

3. OBB based hierarchical collision detection

In this section we give a short outline of hierarchical collision detection based on OBBs.

3.1. Hierarchical collision detection

Hierarchical collision detection use model sub division to accelerate collision queries between two geometric objects by quickly discarding (culling) large subparts of each object that is not in collision with the other.

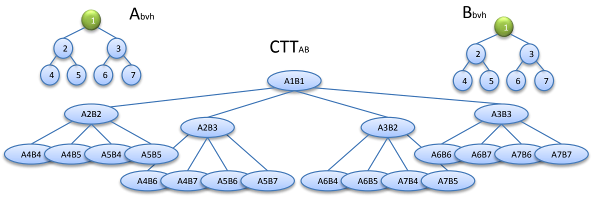

Figure 1: *The resulting bounding volume test tree from traversing the bounding volume hierarchies A_{bvh} and B_{bvh} using a simultaneous descent strategy.*

These data structures are so-called bounding volume hierarchies (BVH). Each node in a BVH has a bounding volume (BV) attached that bounds one or more primitives of the geometry, typically triangles. A child node bounds a subset of the primitives that its parent node bounds.

When determining if two objects are colliding the BVH trees are traversed from the root BV and intersection tests are performed on BV's from each tree. If a node A intersects with another node B, then child nodes of A and B will be checked for intersection. If the traversal reaches two leaf nodes then intersection between the primitives of the nodes are computed. This approach effectively decrease the number of primitive intersection tests compared to a naive $O(N^2)$ approach. Many optimizations of both tree traversal, tree building and tree structure are available. More in depth descriptions can be found in [Eri05].

3.2. The bounding volume test tree

The Bounding Volume Test Tree (BVTT) is the combination of two or more BVH's given some descent strategy, into a tree where a node represents an overlap test between two nodes from different BVH's. A descent strategy determines how to proceed when two nodes from two BVH's are overlapping. The most common descent strategy is the alternating strategy where the children from one of the overlapping parent nodes is tested for overlap with the other parent node. The simultaneous descent strategy descents in both parent nodes and tests the children of one parent node with the children of the other parent node.

Assuming the BVH's are binary trees, a simultaneous descent rule will result in a 4-ary BVTT as shown in figure 1 if an alternating descent rule is used, the BVTT will be a binary tree.

3.3. Efficient intersection test

Intersection tests are the heart of any hierarchical collision detector. In this study we have chosen OBBs as the bounding volume type and triangles to describe the object geometry. Computational complexity of OBB overlap tests are of medium complexity compared to that of spheres and k-dops, and earlier work [GLM96] has shown good results using this type of bounding volume.

The efficiency of an algorithm for intersection tests is largely dependant on the architecture on which it is running. E.g. streaming architectures usually don't handle branch logic efficiently. In previous work [EF08] two algorithms for triangle overlap testing was implemented and benchmarked on the GPU.

- Devillers - This triangle intersection algorithm was presented in [DG02]. It utilise extensive branch instructions to reduce total number of operations and therefore requires effective and complex branch handling.
- Segment-piercing - Straight forward computation of triangle intersection by testing if any segment of a triangle intersects the other triangle. This results in 2x3 segment-triangle intersection tests.

Devillers triangle overlap test is by far the fastest on a modern PC because of the PC's advanced branch handling. But with the lack of good branch handling on the GPU the computationally heavier but simpler Segment-piercing algorithm performs up to 3 times faster.

Testing intersection between OBBs is less complex than between triangles and the method based on the separating axis theorem proposed by Gottschalk [GLM96] is considered very efficient on a standard PC. When running on the GPU though, there is less benefit from the early exits that can be taken when there is no intersection between two OBBs. This is due to the SIMD nature of the GPU which suffers from branching.

3.4. Issues

Collision detection algorithms based on hierarchical bounding volume trees are inherently serial in nature. A node in the BVTT must be checked before any of its children are checked. Therefore it is not obvious how to apply such an algorithm to a massively parallel architectures, e.g. GPUs.

4. Method

The GPU is a massively parallel architecture which can be exploited through the CUDA API. Though CUDA has support for branching and inter-thread communication, a GPU is optimized for processing of large data sets in a SIMD fashion. Serial code can be hard to fit on the architecture and will rarely reach the same instruction throughput as on a general purpose CPU.

It was therefore decided to focus on accelerating the intersection tests of OBBs/triangles along with the matrix multiplications involved in coordinate transformations between object frames. These operations are characterised by high floating-point intensity, small amount of branches and requiring no inter-thread communications.

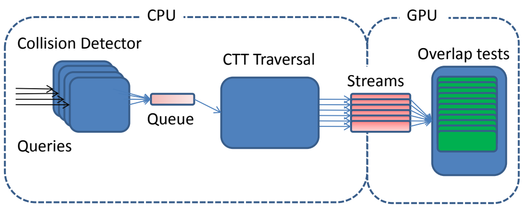

Figure 2: *Multiple threads are using an instance of the collision detector. Queries are traversed in parallel and batches of OBB and triangle tests are gaathered in workpiles. The workpiles are send to the GPU for processing through a number of streams.*

The collision detection algorithm rely on the application to exploit coarse grained parallelism for enabling cost effective utilisation of the GPU architecture. As such, multiple queries to the collision detector are expected to be called in parallel.

Figure 2 show an overview of the hybrid method, that works partly on the CPU and partly on the GPU. A number of threads each has a collision detector instance that defines an interface for querying for collisions on a workcell(virtual environment) with a specific configuration.

All queries are pushed onto a queue from where the BVTT traversal is initiated by a separate thread. Box and triangle overlap tests are assembled into workpiles, by traversing multiple BVTTs which possibly span multiple queries. The workpiles are sent to a GPU manager thread that gathers workpiles from multiple queries and streams them to the GPU for overlap testing. The overlap test results are forwarded back to the traversal thread where new overlap batches are assembled for processing.

In the following the three main stages of the algorithm are described: *tree traversal*, *gathering* and *overlap testing*.

4.1. BVTT traversal

A collision query is composed of multiple collision objects that each require a BVTT traversal state. The state maintains a stack of overlap test batches that are not yet processed. The pseudo code in Algorithm 1 illustrate the flow of the basic collision query.

The workpile is populated by "initial work" which is the set of root nodes of all BVTT's that is being traversed. In applications where there is strong temporal coherence between two consecutive collision queries, initial work can be initialised with the front of the previous query. The front as shown in figure 3 is the list of all nodes in the BVTT that is non-overlapping.

Algorithm 1 The Collision Query

```
   updateObjectTransforms( currentstate )
2: addInitialWork( traverseStates, workpile )
   calculate transform $T_A^B$
4: while !obbBatchStack.empty() do
     GPUWorker.processWork( workpile );
6:   wait( );
     assembleWork( traverseStates, result );
8: end while
```

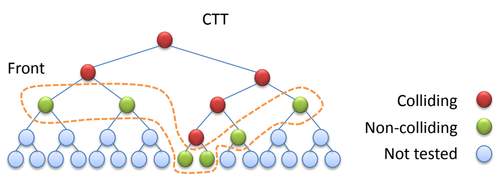

Figure 3: *The front is used to hot start the testing of nodes in the BVTT when temporal coherence can be exploited. In the best case scenario half the test nodes can be culled.*

After the workpile has been populated with initial work it is sent to the GPU where overlap tests are processed. The result is a list of integer values that encode the overlap test results.

Traversing the BVTTs is done using a simultaneous descent strategy where all children of one bv node is tested for overlap against all children of another bv node. This generates more work than the traditional approach where a parent bv node is tested against the children of another bv node. Even though more work is generated the strategy fits well into the GPU architecture because it generates smaller batch descriptions and therefore reduce bandwidth requirements between CPU and GPU. The strategy also benefits from faster descending into the BVTT.

To reduce the overhead of traversing the BVTT an implicit pure balanced node tree is used. This means that no fetching of memory is necessary to get the child indexes of a node, or to test if a node is a leaf or not. The traversal is therefore quite fast and uses only little memory, but it requires a perfectly balanced tree.

To generate a perfectly balanced tree a top-down construction using a median splitting strategy is used. This results in a nearly balanced tree where only the last level in the tree is incomplete. To populate the last level in the tree all leaf nodes in the level above the last level are copied into their children thereby creating a perfectly balanced tree. This however generates redundant nodes in the last level of the tree and in worst case half of all the bounding volumes become redundant.

4.2. Gathering/processing of WorkPiles

When the BVTT has performed a descent step of the front and populated a workpile, it is sent to the GPU for processing. The batches contained in the workpile are copied to the global device memory before they can be processed. The collision results are copied back to the host when the GPU has finished processing the workpile.

In order to reduce the overhead of these memory transfers, the asynchronous streaming API of CUDA is utilized. The processing of WorkPiles is also a separate thread from the BVTT traversal. The BVTT traversal may therefore continue while the GPU is processing workpiles, as long as enough independent work in the scene is provided.

Periodically the thread managing the GPU will poll the running streams for completion, which indicates that the GPU has finished copying the intersection test results back to the host. The thread then notifies the BVTT traversal stage, and continues.

4.3. Overlap testing

The overlap testing kernels are started after the host has finished copying the batches to the global device memory. The different types are treated separately, that is triangles and OBB batches are processed in separate kernel launches.

Inside the kernels the mapping of batches to CUDA kernel threads is 1:1, i.e. a single thread performs the tests contained in a single batch. For moderate to large batch sizes this simple scheme performs well. During initial partitioning of the problem, the system will try to maximise the number of threads running within a thread block. Should there be more batches than the maximum number of threads within a thread block, more thread blocks will be launched. As expected the performance of the GPU will only be high when there are enough thread blocks to occupy the multi-processors. For the GTX 260 used in the experiments this is around 24 to 30 blocks. (An even larger number of thread blocks is in theory helpful for hiding latency, but this has not been observed.)

The steps within the OBB kernel can be seen in Algorithm 2. The triangle kernel is very similar to the OBB kernel, except that it requires other transformations and algorithms for overlap testing.

For overlap testing of batches, the GPU requires all primitives (OBBs, triangles) and object-object transformations to be present in the device memory. The primitives do not change during a simulation and are thus only transfered once in a simulation run. The transformations on the other hand may change with each new query. The transformations are kept in a portion of the GPU device memory which is cached (constant memory).

The upper bound on coarse-grained parallelism relates to the number of queries which can be called in parallel. As

Algorithm 2 OBB Overlap Kernel

```
    fetch batch from device memory
 2: calculate OBB indices from batch
    calculate transform $T_A^B$
 4: for all $obb_A \in$ OBBs to be tested from object $A$ do
        calculate $T_{obb_A}^B$
 6:     for all $obb_B \in$ OBBs to be tested from object $B$ do
            calculate $T_{obb_A}^{obb_B}$
 8:         res $\leftarrow overlapTest(obb_A, obb_B, T_{obb_A}^{obb_B})$
        end for
10: end for
    write res to device memory
```

each of these parallel queries require a set of unique transformations, there needs to be at least room for $N \times M$ transformations, where N is the number of parallel queries and M is the number of objects in the scene. For the current implementation the limit is due to the location in constant memory.

As expected for the massively parallel GPU, there needs to be several thousand batches for it to reach peak performance. This is due to the large number of threads, the launch overhead of the kernels and the scheduling system within the GPU. Achieving enough batches is done by either having a large scene with many objects or running multiple queries of the scene in different configurations in parallel.

5. Performance model

The cost of a proximity query when using bounding volume hierarchies is usually calculated with the following expression:

$$T = N_{bv}C_{bv} + N_pC_p \qquad (1)$$

where T is the total cost, N_{bv} is the number of bounding volume pair tests, and C_{bv} is the total cost of a bounding volume pair test including the cost of transforming one bounding volume into the bounding volume coordinate frame. N_p is the number of primitives tested for collision and C_p is the cost of testing a pair of primitives for overlap.

As with the traditional cost function we expect the cost of our algorithm to depend on the number of OBB and triangle pair tests that is performed. Though since our algorithm basically runs in multiple threads the function needs to be modified. We have to consider two cost functions namely that covering traversing the BVTT and that which covers processing of workpiles.

The cost of traversing the BVTT is usually not explicitly modelled in the traditional cost function. Though it should be obvious that the cost of traversing the BVTT is closely related to the number of OBB and triangle pair tests that are performed. So we use an expression similar to that of the traditional cost function:

$$T_{cpu} = N_{bv}C_{t,bv} + N_pC_{t,p} \qquad (2)$$

where $C_{t,bv}$ is the cost of traversing one OBB test pair and $C_{t,p}$ is the cost of traversing one triangle pair.

When processing workpiles the GPU processes OBB and triangle pair tests in seperate workpiles. The cost function of the intersection tests and memory transfers in itself is similar to the traditional cost function, though the initialisation of kernels and memory copies gives rise to an offset $C_{p,off}$ and $C_{bv,off}$ in the cost function. Since workpiles are processed as either OBB pair or triangle pair workpiles two identical cost functions can be derived.

$$\begin{aligned} T_{bv,GPU} &= \sum_{i=0}^{N_{bv,wp}} C_{bv,off} + C_{bv}N_{bv,i} && (3) \\ &\approx N_{bv,wp}(C_{bv,off} + C_{bv}N_{bv,avg}) && (4) \end{aligned}$$

where $N_{bv,wp}$ is the number of OBB workpiles and $N_{bv,avg}$ is the average number of OBB batches in a workpile. The cost function of the triangle processing is similar

$$T_{p,GPU} \approx N_{p,wp}(C_{p,off} + C_pN_{p,avg}) \qquad (5)$$

6. Test Results

In this section we present benchmark results of our method. The collision detection library PQP running on a single thread was used as reference. To enable a fair comparison when using multiple parallel queries in the tests we forced our algorithm to execute on a single cpu core.

All tests were run from a PC with an Intel Core 2 Quad 2.83GHz processor and 2GB DDR2 memory, running GNU/Linux (Ubuntu 8.10). A NVIDIA GeForce GTX 260 graphics card was used as the platform for the GPU benchmark, used in the same PC previously mentioned. The algorithms were implemented in CUDA 2.2 with single-precision floating point.

Three different classes of tests were performed:

- BVTT traversal performance - Though normally not taken into consideration when discussing performance of BVH methods we have experienced that the time taken to traverse the BVTT is not insignificant. These tests will help determine an upper bound on the expected speedup when considering the CPU as the bottleneck.
- Processing performance - the performance of the GPU for overlap tests are used to estimate an upper bound on the expected speedup assuming the GPU to be the performance bottleneck. The tests include both box-box and triangle-triangle overlap tests.

- Collision detection in simulation - the actual performance of the complete algorithm when used for rigid body simulation of a fairly complex scene.

In the following tests the triangle-triangle intersection algorithm on the GPU is not capable of handling co-planarity. The co-planarity can be discarded if we assume that models are solid models and if two equal models never have the exact same configuration.

6.1. Test data

Triangle and box overlap tests are performance wise sensitive to whether the test returns true or false. In the box-box overlap test based on separating axis a non-colliding box-pair can be up to 15 times faster than a colliding one. So to enable a realistic performance evaluation the collision test data is constructed from a full simulation scenario.

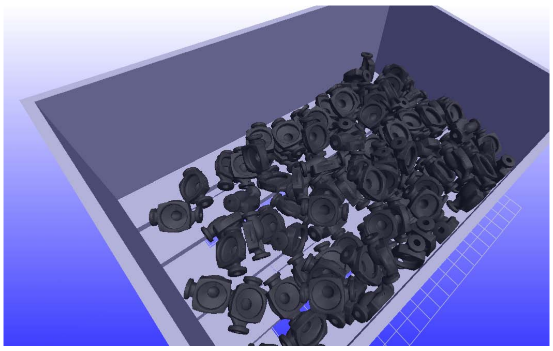

Figure 4: *Scene used for generating realistic contact data. Each of the 106 objects has more than 4000 triangles.*

In figure 4 a large number of fairly complex objects are dropped into a pallet. The setup is used to generate realistically looking pallets of objects for training a vision system in a bin picking application. The simulation span over 700 time steps where each timestep require collision detection. In the first few frames of the simulation all objects are disjoint and the collision detector is therefore only burdened with a small number of overlap tests. As the simulation progresses the number of colliding objects increase drastically and so does the load on the collision detector.

6.2. BVTT traversal performance

Traversing the BVTT is expected to be low cost compared to doing actual overlap tests. Though since we run accelerated overlap queries in parallel the cost of BVTT traversal is important to the actual speedup. E.g. if BVTT traversal takes 10% of the time then, as a consequence of Amdahl's Law, we will never get a speedup of more than a factor 10. The purpose of this test is therefore to investigate the cost of traversing the BVTT.

The total traverse time depends on the number of visited nodes in traversing the BVTT. In our implementation the traversing is split in an assemble and an update step. Figure 5 show the combined assemble and update performance in respect to the number of batches in a workpile. Performance results for both OBB workpiles and triangle workpiles are plottet and lines are fitted to the data such that we estimate $C_{t,bv} = 6.76 \cdot 10^{-5} ms/batch$ and $C_{t,p} = 2.93 \cdot 10^{-5} ms/batch$.

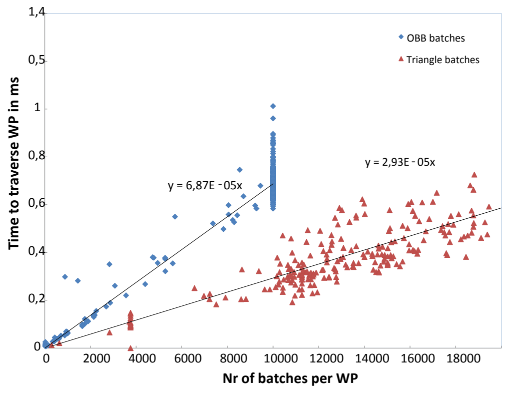

Figure 5: *Cost of traversing the BVTT with respect to the nr of batches in a workpile that is generated from the traversing.*

6.3. Processing performance

The processing timing includes copying workpile to the GPU, testing the OBB or triangle batches on the GPU and copying results back from GPU. Figure 6 show two data series: one for workpiles of OBB batches and one for workpiles of triangle batches. Using the fitted tendency lines we estimate $C_{bv,off} = 2.36 \cdot 10^{-5} ms/batch$ and $C_{bv} = 9.61 \cdot 10^{-2}$ as well as $C_{p,off} = 1.52 \cdot 10^{-5} ms/batch$ and $C_p = 6.21 \cdot 10^{-2}$.

The relative large offset for both OBB and triangle workpiles indicate that large workpiles is needed to increase performance. Consider using workpiles with an average size of 5000 batches. In such a case only half the processing time is used for actual processing of overlap queries.

6.4. Collision detection in simulation

Simulation is sequential in nature, to process the current frame the results of the previous frame is needed. So for small simple scenes we do not expect the GPU accelerated algorithm (PCD) to perform much better than the CPU based algorithm.

In this test the scene is fairly complicated though the number of colliding objects change over time. Figure 7 show the

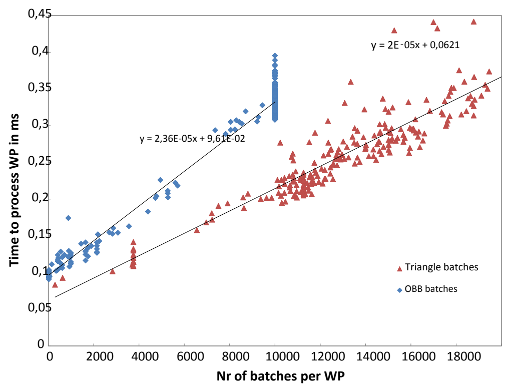

Figure 6: *Cost of processing OBB and triangle batches with respect to the nr of batches in a workpile.*

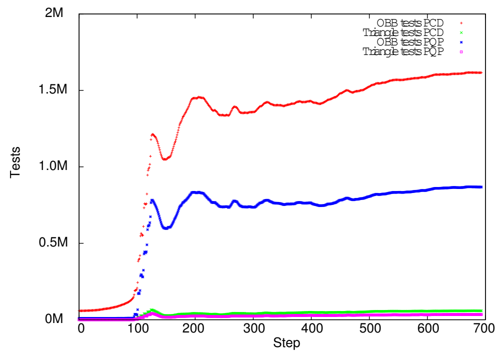

Figure 7: *Nr of OBB and triangle tests in each time step for both PQP and PCD.*

number of OBB and triangle overlap tests that are performed in each time step. In general PCD performs about twice the amount of OBB tests and triangle tests than that of PQP. In the beginning of the simulation where no objects are colliding PCD even performs up to 10 times the amount of work than PQP. This increased amount of work is a result of the simultaneous descent method and the less efficient BVH structure that is used in PCD.

Though more work is performed the overall performance of PCD is clearly better than that of PQP. Figure 8 show the query time in each time step of the simulation. In the first 100 steps very few overlap tests are performed and PCD is slightly slower than PQP. The number of overlap test increases drastically after the 100th step and the performance of PCD increase to about 6 times that of PQP.

In table 1 the total time for running the simulation sequence with the different methods is listed. The PCD-P is PCD where individual time frames are executed in parallel. Parallel execution only makes sense in applications that run multiple simulations.

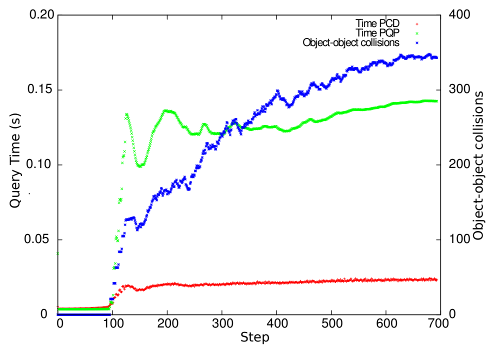

Figure 8: *Time to perform complete collision queries in each time step of the simulation.*

	PQP	PCD-S	PCD-P
Time	75.692s	12.9801s	10.9
Gain	1	5.83	6.94

Table 1: *Performance comparison.*

7. Discussion

It is obvious that the average workpile size has a large impact on the efficiency of the processing step due to the relatively large offsets $C_{bv,off}$ and $C_{p,off}$. By investigating both traverse and processing performance we can calculate the average workpile size where the cost of traversing or processing is equal. We calculate the intersection of the tendency lines and get:

$$N_{bv,avg} = \frac{C_{bv,off}}{C_{t,bv} - C_{bv}} \approx 2130 \tag{6}$$

$$N_{p,avg} = \frac{C_{p,off}}{C_{t,p} - C_p} \approx 4404 \tag{7}$$

Which basically means that for $N_{bv,avgwp} > 2130$ or $N_{p,avgwp} > 4404$ the BVTT traversal will become the performance bottleneck of the algorithm. For such a small workpile size the processing step is highly inefficient, see figure 6. This suggests that optimizations to reduce the cost of the BVTT traversal will result in an overall performance gain.

The simultaneous descent method was chosen because it reduced both the load on traversing and communication to the GPU. This admittedly also doubled the amount of overlap tests but the increased number of overlap tests had no effect on the overall performance since the BVTT traversing was identified as the bottleneck.

8. Conclusion and future work

We presented a hybrid hierarchical based collision detection method where BVTT traversal was implemented on the CPU and overlap testing of OBB and triangles was implemented on the GPU.

To reduce the cost of traversal we modified the original algorithm with: a pure implicit tree structure, a simultanous descent strategy and created bounding volume hierarchies with up to 2 triangles in each leaf. Though these modifications led to a doubling in overlap tests an overall performance gain of a factor 7 was still observed.

To increase the performance even further the BVTT traversal should be optimised. Using k-dop or other more complex bounding volumes instead of OBB would create smaller bounding volume tree and as a result the traversal should be faster.

Another approach would be to move more responsibility to the GPU. Instead of only testing one BVTT node per thread each thread could traverse a smaller sub tree of the BVTT. This however would possibly add complex synchronisation and branching code to the GPU kernels which has a negative impact on the performance.

References

[BW03] Baciu G., Wong W. S. K.: Image-based techniques in a hybrid collision detector. *IEEE Transactions on Visualization and Computer Graphics 9*, 2 (2003), 254–271. 2

[BWS98] Baciu G., Wong W., Sun H.: Recode: An image-based collision detection algorithm. *Computer Graphics and Applications, Pacific Conference on 0* (1998), 125. 2

[CLMP95] Cohen J. D., Lin M. C., Manocha D., Ponamgi M. K.: I-collide: An interactive and exact collision detection system for large-scale environments. In *In Proc. of ACM Interactive 3D Graphics Conference* (1995), pp. 189–196. 1

[DG02] Devillers O., Guigue P.: *Faster Triangle-Triangle Intersection Tests*. Research Report 4488, INRIA, 2002. 1, 3

[EF08] Ellehoej T. F. K., Fugl A. R.: *Acceleration of Collision Detection on Parallel Computer Architectures*. Master's thesis, University of Southern Denmark, 2008. 3

[Eri05] Ericson C.: *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Morgan Kaufmann, January 2005. 2

[GLM96] Gottschalk S., Lin M. C., Manocha D.: Obbtree: a hierarchical structure for rapid interference detection. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM, pp. 171–180. 1, 3

[Hel97] Held M.: Erit – a collection of efficient and reliable intersection tests. *Journal of Graphics Tools 2* (1997), 25–44. 1

[HTG04] Heidelberger B., Teschner M., Gross M.: Detection of collisions and self-collisions using image-space techniques. In *Journal of WSCG* (2004), pp. 145–152. 2

[KHH*09] Kim D.-S., Heo J.-P., Huh J., Kim J., eui Yoon S.: HPCCD: Hybrid parallel continuous collision detection using cpus and gpus. *Computer Graphics Forum (Pacific Graphics) 28*, 7 (2009). 2

[KHM*98] Klosowski J. T., Held M., Mitchell J. S. B., Sowizral H., Zikan K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics 4* (1998), 21–36. 1

[LMM10] Lauterbach C., Mo Q., Manocha D.: gproximity: Hierarchical gpu-based operations for collision and distance queries. In *Computer Graphics Forum (Proc. of Eurographics)* (2010). 2

[NA05] Nuzhet Atay John W. Lockwood B. B.: A collision detection chip on reconfigurable hardware. In *13th Pacific Conference on Computer Graphics and Applications* (Oct. 12-14 2005). short paper. 2

[NA06] Nuzhet Atay B. B.: A motion planning processor on reconfigurable hardware. In *International Conference on Robotics and Automation* (May 2006), pp. 124–132. 2

[Ngu07] Nguyen H.: *Gpu gems 3*. Addison-Wesley Professional, 2007. 2

[OLG*07] Owens J. D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A. E., Purcell T. J.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113. 2

[RHZA06a] Raabe A., Hochgürtel S., Zachmann G., Anlauf J. K.: Hardware-accelerated collision detection using bounded-error fixed-point arithmetic. In *Proceedings of WSCG 2006* (Plzen, Czech Republic, 30 January–3 February 2006), Skala V., (Ed.), Union Agency, pp. 17–24. 2

[RHZA06b] Raabe A., Hochgürtel S., Zachmann G., Anlauf J. K.: Space-efficient FPGA-accelerated collision detection for virtual prototyping. In *Design Automation and Test in Europe (DATE)* (Munich, Germany, 6–10 March 2006). 2

[TTSD06] Tropp O., Tal A., Shimshoni I., Dobkin D. P.: Temporal coherence in bounding volume hierarchies for collision detection. *International Journal of Shape Modeling 12* (2006), 159–178. 1

[YM06] Yoon S.-E., Manocha D.: Cache-effcient layouts of bounding volume hierarchies. *Computer graphics forum (Eurographics) 25* (2006), 506–516. 2