Fast Hierarchy Operations on GPU Architectures

Abstract

We present fast parallel algorithms to construct, update and traverse bounding volume hierarchies on many-core GPUs. Our approach is general and based on a new work distribution scheme that can easily adapt to dynamically changing work loads that arise in interactive applications. This enables us to exploit all cores to perform hierarchical operations using tight-fitting OBBs with little overhead as compared to widely-used AABBs. We use our fast hierarchy refitting for interactive ray tracing and collision detection on complex and deforming models such as the problem of self-collision. In practice, we observe more than an order of magnitude improvement in the performance of continuous collision queries as compared to prior GPU-based algorithms, competitive or better performance compared to multi-core CPU methods, and much higher speed for hierarchy refitting with applications to ray tracing.

1 Introduction

Hierarchical data structures and algorithms are widely used in computer graphics and related areas. They are used to implement some of the basic operations and accelerate rendering or simulations on complex data sets. Some of the most widely used applications include ray tracing, collision and proximity queries, visibility computations, scientific computations, database computations and physically-based simulations.

There is extensive literature on hierarchical data structures and algorithms. These hierarchies can be classified into spatial partitioning hierarchies (e.g. kd-trees, BSP trees, octrees, etc) and bounding- volume hierarchies (e.g. sphere trees, AABB trees, OBBtree, etc). At a broad level, the underlying computations on these hierarchies can be classified into three types of operations:

1. **Hierarchy construction:** Given a set of primitives, many top-down or bottom-up techniques have been developed to construct or compute a hierarchy.
2. **Hierarchy update:** As some of the primitives change or undergo motion, the hierarchies are updated accordingly.
3. **Hierarchical traversal:** The nodes of the hierarchies are traversed in some order such as depth- or breadth-first.

In the rest of the paper, we will primarily focus on efficiently performing these operations on GPUs for bounding volume hierarchies (BVHs), but the underlying techniques are also applicable to spatial partitioning hierarchies.

One of our goals is to exploit the parallel capabilities of current many-core GPUs to efficiently perform these hierarchy computations. Some of these operations involve a high number of memory accesses and thereby result in a reduced computational intensity of the overall algorithm. Moreover, the workload of the resulting algorithm can vary considerably between successive frames of a simulation. For example, the cost of hierarchy traversal in a collision detection algorithm depends on the relative configuration of the two moving objects. Therefore, it can be challenging to utilize all the computational resources to provide coordination for distributing the dynamically created workload. The latter is a problem in particular since there is no direct intercommunication channel between the GPU cores. In addition, memory accesses have high latency which can add considerable overhead if coordination between the cores is performed using main memory.

Overall, we categorize operations on hierarchies as trivially and non-trivially parallelizable on current GPUs. For example, most search operations that traverse hierarchies do not pose a challenge because multiple queries can be processed independently and thus are straightforward to implement using both thread and data parallelism. Examples include ray tracing [Purcell et al. 2002], spatial hashing [Lefebvre and Hoppe 2006] and similar approaches that can easily traverse the hierarchies in parallel on GPUs. A greater challenge is posed by algorithms that either change or update the hierarchies between frames or cannot be easily decomposed into independent tasks. These include building a hierarchy as well as search operations that do not start with independent queries and can only be subdivided into parallel operations during run-time traversal.

In this paper, we present new parallelization algorithms for efficiently processing several of the non-trivial hierarchy operations on current massively parallel GPU architectures. In practice, these hierarchy operations have the following aspects:

- The degree of available parallelism changes rapidly.
- The work load is not known in advance and can shift dynamically during the execution.

Thus, a significant degree of coordination between cores is required in the algorithm to maintain parallelism. Examples of these operations include construction and refitting of object hierarchies as well as simultaneously traversing two hierarchies to find the overlaps. Prior work for these algorithms on GPUs had to use explicitly maintained work queues which introduces a high overhead and forced synchronization between each processing step. In practice, this only works for applications where each individual step is long enough to mitigate the maintenance cost.

Instead, our approach is based on explicit balancing of work units coupled with very lightweight synchronization between cores to communicate the need for load balancing. This improves on previous methods by distributing the workload only when absolutely necessary and thus minimizes synchronization overhead. Moreover, it makes it possible to apply our approach to very fine-grained workloads such as those that arise as part of discrete and continuous collision detection. Our system provides an order of magnitude improvement compared to previous GPU algorithms for collision detection and is even faster than the fastest CPU-based single-core and multi-core hierarchy algorithms. In addition, we demonstrate that the high computational power on GPUs allows us to use tighter fitting bounding volumes, such as oriented bounding boxes (OBBs), at a very small overhead compared to standard axis-aligned bounding boxes (AABBs), resoluting in a overall speedup. This is in contrast to most recent CPU-based collision detection algorithms that tend to use simple bounding volumes such as AABBs for collision and proximity computations. Furthermore, for deformable models our hierarchy refitting algorithm is an order of magnitude faster than previous CPU implementations, which allows faster speed for applications such as collision detection and ray tracing of dynamic scenes. Finally, we demonstrate applications to hierarchy construction with a speedup of up to 15% due to better load balancing.

2 Background

We first briefly summarize previous work in the areas of GPU work organization, parallel proximity queries and collision detection as well as hierarchy construction. We then provide some background on current GPU architectures and their main defining features.

2.1 Previous work

There is extensive literature on hierarchical data structures and algorithms as well as GPU computing. Many GPU-based algorithms exploit hierarchies for fast rendering or simulations. These include visibility computations [Goradia et al. 2008], photon mapping [Purcell et al. 2003], fluid simulation [Kyle Hegeman and Miller 2006], NURBS rendering [Guthe et al. 2005], distance transforms [Cuntz and Kolb 2007], global illumination [Szirmay-Kalos et al. 2008], adaptive shadow maps [Lefohn et al. 2007], random access data structures [Lefohn et al. 2006], spatial hashing [Lefebvre and Hoppe 2006] and random access trees [Lefebvre and Hoppe 2007]. Some of these applications used specialized hierarchy operations or performed specific hierarchical operations.

Many techniques have been proposed for fast GPU-based ray tracing. These methods either use kd-trees [Foley and Sugerman 2005; Horn et al. 2007; Zhou et al. 2008] or BVHs [Lauterbach et al. 2009] and traverse each ray in parallel without any coordination between them. Hierarchical techniques based on BVHs are widely used to accelerate collision and proximity queries. These include sphere trees, AABB trees, OBBTrees, K-DOP trees, etc. [Ericson 2004]. However, there is a widespread notion that it is hard to implement these hierarchical algorithms on GPUs. Rather, most GPU-based collision checking algorithms exploit the rasterization capabilities of GPUs by using depth or stencil buffer tests [Heidelberger et al. 2003; Knott and Pai 2003; Govindaraju et al. 2003]. In order to handle complex models, the hierarchical traversals are performed on the CPUs [Govindaraju et al. 2005; Sud et al. 2006] and this results in additional CPU-GPU overhead. Other GPU-based algorithms are used for broad-phase collision only [Le Grand 2007].

There is considerable literature in parallel computing on the use of work queues for load balancing, including locking and non-locking shared queues such as work stealing approaches [Arora et al. 1998; Hendler and Shavit 2002; Chase and Lev 2005; Hendler et al. 2006]. These techniques map very well to hierarchical and recursive operations and have been employed extensively in parallel systems and parallel programming languages such as Cilk [Frigo et al. 1998]. However, they have not been used on GPUs as the overhead of performing communication between the cores through main memory is high. Instead, previous techniques for GPU work queues used explicit compaction methods between kernel calls. The overhead of these methods makes them efficient only for applications with relatively high computational intensity and coarse-grained parallelism [Zhou et al. 2008; Lauterbach et al. 2009]. There are known parallel algorithms to accelerate hierarchical traversals [Rao and Kumar 1987; Kumar and Grama 1994] and they are also applied to parallel collision detection [Kitamura et al. 1995; Grinberg and Wiseman 2007]. However, most of these methods either perform communication between the processors or are not suited for GPU-like architectures.

2.2 GPU architectures

In this section, we give a brief overview of current many-core GPU architectures that we exploit in our algorithm. In general, GPUs have a relatively large number of independent processing cores, each of which is optimized to perform vector operations but runs at comparatively low frequencies compared to current CPU architectures. The high vector width – between 8 and 64 for current generation of GPUs – also implies that any efficient algorithm needs to utilize data parallelism to achieve high performance. Another main issue is the GPU memory system that typically provide more bandwidth as compared to CPU memory systems, but has a higher latency. Moreover, the caches in GPUs are much smaller than CPU caches and current GPUs only provide read-only access, which limits their use for general purpose computing. The main goal of these caches is not to reduce latency, but rather to reduce the amount of memory bandwidth used.

In order to improve latency, GPUs use two main techniques: first, each core has local scratch memory that can be used in programs as an explicitly managed store and provides very low latency. Thus, if algorithms are designed with sufficient locality, most of their memory accesses should go there. Second, the high main memory latency is circumvented by use of hardware multi-threading while waiting for the results of memory accesses. This can be achieved by running several data-parallel tasks on each core such that the processor can switch between them as they are blocked waiting for memory accesses. Finally, the vector size used in programs can be extended beyond the size of the actual hardware vector units such that the computations can be strip mined (i.e. processed in chunks of real vector size). This means that GPUs can efficiently process these large vectors for latency hiding as well. Overall, this means that algorithm design should aim at providing both as much task and as much vector parallelism as possible since it will improve the efficiency of the processors.

3 Work distribution on GPU architectures

In this section, we present our main work scheduling algorithm for GPU architectures that is particularly suitable for hierarchies and other recursive algorithms where load balancing is necessary. Our approach is general and applicable to different GPU architectures. We first introduce the notation used in the rest of the paper and than describe our approach

3.1 Notation

Object hierarchies are represented as a tree of nodes that have pointers to their child nodes and, potentially, their parents in the hierarchy. The leaf nodes instead have references to one or more geometric primitives (e.g. triangles). Each node also includes a bounding volume (BV) such as a sphere, AABB (axis-aligned bounding box), OBB (oriented bounding box), etc. that by definition contains all the children's BVs. In general, we refer to each independent processing unit on the GPU as a core, each of which is a vector processor that executes the same instruction on a data array elements in parallel. A running program is a task that can be executed in parallel on each core of the GPU, and we refer to the specific code run by the task as a kernel that is identical on all the cores. We assume that our kernels take work units as input that can be processed independently and can lead to generation of new work units. The exact definition of a work unit depends on the kernels: for example, when testing two hierarchy nodes for overlap, the result can be that similar intersection tests needs to be performed recursively on the children of these nodes. In this case, the intersection test is the kernel and the work unit is the pair of hierarchy nodes. Overall, we look at the complete set of active work units at any step in the algorithm as the *front*, which may or may not be available in an explicit form. If the order in which hierarchy operations are generated and executed can be represented as a tree, then the front is simply a cut through the tree. The size of the front governs the available parallelism and thus should be as large as possible since elements in the front can be executed independently. In practice, the size depends on the order of execution of work units, the configuration of input objects and the size of the hierarchy.

3.2 Hierarchy workloads

The main challenge when performing parallel operations on hierarchies is the dynamic nature of work distribution. Since the workload is not known a priori, assigning work units to different cores and vector lanes in advance is impossible. A front of sufficient size

to occupy all cores may not even exist until after some steps in the computation. For example, in hierarchical collision detection the initial front just has one element, i.e. the pair of root nodes; as the traversal generates more pairs of nodes the size of the front typically increases in geometric progression. Work is typically also not evenly distributed over the hierarchy since some sub-trees of the hierarchy may be skipped early whereas others need to be processed much deeper. Therefore, some mechanism for distributing work between cores and load balancing during the hierarchy operation is necessary. On multi-threaded CPU architectures, prior approaches have used work queues and work stealing for operations with hierarchies and recursion with similar properties [Arora et al. 1998].

However, these techniques do not currently work well on GPUs for multiple reasons. Primarily, they are based on the assumption that low-latency communication between cores is possible in order to manage concurrent access to shared structures. Unfortunately, this is only possible in a very restricted sense on current GPUs. The main barrier to communication is the latency and lack of a memory consistency model in the global GPU memory shared by the cores, i.e. different cores are not guaranteed to see memory writes from other cores or may not even see them in the same order they were written. Even though newer GPU architectures provide atomic operations such as compare-and-swap (CAS) that could be used for locking operations, the remaining problem is that previous writes to the memory protected by the lock may not have been executed yet, thus preventing implementation of work queues or other structures shared by all cores. Even if memory consistency was not a problem, busy waiting such as by spinning on a lock variable is relatively inefficient on an architecture with high memory latency and hardware multi-threaded execution can also lead to priority inversion and prevent other threads on the same core from performing useful work. In addition, actual task scheduling to the cores is still handled by fixed-function units on hardware that cannot be affected by the programming interface. The number of actual tasks is almost always higher than the number of cores to allow hardware multi-threading, and since the scheduler does not make any guarantees as to fairness in core allocation, it is not guaranteed that any task is actually executed in parallel to any other. Thus, global communication between all tasks cannot be guaranteed as some may not even be started until others end.

Previous approaches that worked with hierarchies such as construction [Zhou et al. 2008; Lauterbach et al. 2009] have adopted a work management model with no communication where synchronization is achieved by running the actual work kernel repeatedly for just one step and work queues are maintained by a separate kernel between work kernel calls. Since the number of actual new work units produced by each step of the work kernel is not known in advance, work lists had to be allocated conservatively such that memory writes could occur without possibility of conflicts. In the context of hierarchy construction, this approach works due to two reasons: a) the work kernel performs a relatively large amount of work per step and b) it could only write out a very small number (two) of new work units per step, thus making list maintenance relatively cheap.

However, for our more general purposes this work organization is not as efficient. In particular, we have work loads where individual kernel calls are far less computationally intensive, so kernel call and work list maintenance overhead as well as latency would become a limiting factor in overall computation. In addition, we are interested in algorithms that also use the data parallel vector units on the individual cores to work on different work units, which means that the amount of work units read and generated during the execution of a task is vastly higher than in simple applications. Finally, the static task allocation performed by the GPU's scheduler means that there can be significant load imbalance if some of the cores are idle while others are still finishing a task. While this may be negligible

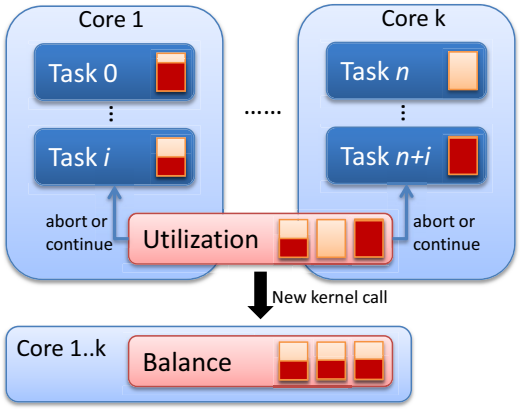

Figure 1: Our approach: *In our approach, each task keeps its own local work queue in local memory and can generate new work units without coordinating with others. After processing a work unit, each task is either able to run further or has an empty or completely full work queue and wants to abort. By checking how many tasks are marked idle after each step, tasks will abort whenever a certain threshold of idle tasks is reached. At that point, an explicit work balancing kernel is launched that rearranges the work queues and distributes work units such that all cores have roughly the same workload.*

for individual steps, these overheads add up if the work task is run repeatedly.

3.3 Our approach

Our approach is motivated by this challenge and tries to circumvent the lack of a memory consistency model and still provide some limited coordination between cores to avoid maintenance overhead. While some explicit queue maintenance work outside of the work kernel is necessary, we drastically reduce the amount of time spent on it by only performing it as a load balancing step between cores when we detect that enough cores are idle. In our work organization approach each task maintains a private work queue either in shared or global memory (depending on the size of work units) that can be read in a data parallel manner to provide work for all vector elements. Putting new elements in the local work queue can be implemented easily by either using a parallel prefix sum based on the results of each work kernel, or atomic increments to the local work queue index to avoid write conflicts. There are two cases where work processing must end: first, the queue is empty and no more work is available and second, the allocated space for the queue is full and the work kernel cannot be executed because any further work elements could not be stored. In that case, we consider the task inactive and atomically increment a global idle counter. To make sure that a sufficient amount of tasks is active to ensure parallel utilization of the whole GPU, each task reads the counter after processing a work unit and compares against an idle threshold to determine whether re-balancing work is needed. If that is the case, then the task aborts (see Fig. 1 for illustration.) Note that even without a memory consistency model, every task will see increments to the global counter eventually.

In case balancing is needed, a we execture a global work distribution kernel that steals work from some of the queues and distributes it to those that are not full. Work redistribution is performed in parallel by first counting the total number of work items and computing a roughly equal number of work units for each queue. Resorting the work units to the new queues for all cores is performed in parallel by computing offsets through a prefix sum algorithm and

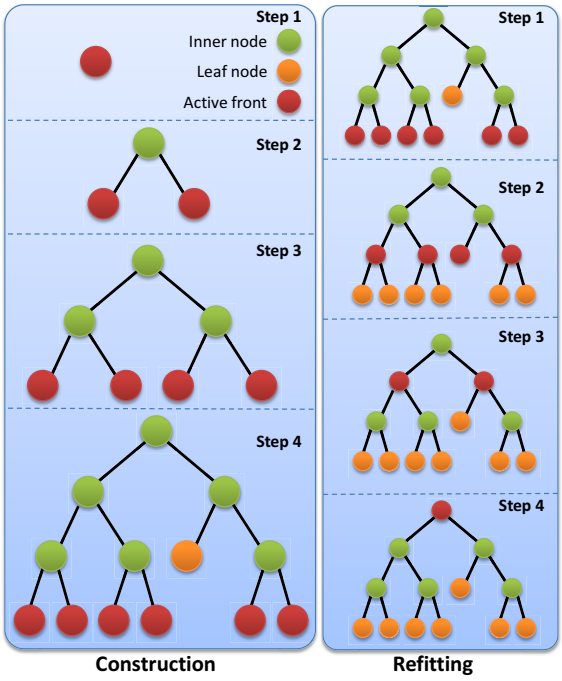

Figure 2: Hierarchy construction and update: *The front of active work units during several steps in hierarchy construction (left) and refitting (right). Each entry in the front can be processed independently by parallel tasks, but subsequent steps depend on output from the previous one and thus require some degree of coordination. We try to maximize the size of the front in order to increase available parallelism.*

then copying to the new positions by all cores in parallel. Overall, there are a couple of factors influencing the performance of this approach, most of which are dependent on which actual work kernel is used. Foremost, the idle threshold should be set relatively high to avoid having to balance work too often, which incurs extra overhead for multiple kernel launches and synchronization overhead. In addition, the number of tasks launched in parallel should depend roughly on the number of cores on the GPU, taking into account that each can run multiple tasks thanks to hardware multi-threading. Therefore, even if a part of the tasks is idle, others running on the same core may still be active.

4 Hierarchical Computations

We now discuss several hierarchical algorithms that are used in existing applications, and how to efficiently implement them on a GPU architecture using our work distribution approach. We show how these algorithms differ both in their traversal as well as in their work creation pattern. Section 6.2 then evaluates the choices for bounding volumes in the hierarchy and how they affect the overall performance.

4.1 Hierarchy maintenance and construction

There are two main operations that modify the actual hierarchies used in all the applications presented in this paper, hierarchy *construction* and *refitting*.

Construction: The construction algorithm processes a set of geometric primitives (e.g. triangles) and builds a hierarchy on them without any knowledge as to the structure of the scene they represent. In general, top-down construction methods are most frequently used for dynamic or deforming models and work by recursively splitting the set of primitives in several subgroups depending on the desired branching factor of the tree (e.g. two for binary trees). Since these splits usually do not result in the same amount of primitives in all the subgroups, the height of the tree is not known in advance and the tree in general can be highly unbalanced. Thus, load balancing is critical to good performance in many applications.

Previous approaches for hierarchy building on GPUs [Zhou et al. 2008; Lauterbach et al. 2009] have implemented the top-down split approach with work queues where each split can either create two new ones or terminate by creating a leaf in the tree (see Fig. 2). Since all splits can be handled totally independently, this approach maps easily to our method. We assume a branching factor of two for better comparison. Thus, since the split operation itself can be implemented as a data parallel operation, only one split is performed per task and this creates up to two new work units in the local queue. The split task is relatively heavy-weight and therefore we perform an idle check after each run and balance the work load as necessary. During the start of the algorithm, a balance will be necessary after each step since the parallelism increases geometrically and starts with only one active split. As soon as enough splits are available to fully utilize the hardware, termination and re-balancing is only necessary for load balancing.

Refitting: The hierarchy refitting or updating operation assumes that a hierarchy already exists, but that primitives may have been moved or deformed, and it will modify the bounding volume associated with each node of the hierarchy to reflect the newly changed primitives. In this case, the primitives associated with each node of the tree do not change and the work structure is simple since it simply needs to perform a post-order traversal of the hierarchy. At each step, the main work is updating the bounding volume of each node based on the bounding volumesof the children. At the leaf nodes, the bounding volumes are computed from the actual geometric primitives. In a normal recursive implementation, the traversal thus goes down the tree to the leaves, then propagates the new bounding volumes upwards to be merged until the root is reached. It is possible to express this in our framework by issuing two different tasks, a downward traversal and an upward merge task. However, since the actual computational load is already very small, we found that it is faster to reorder the layout of the tree in memory such that the nodes are stored by tree level. Thus, we start out with the front at the leaves of the tree and only need to perform the merge kernel. Since each level is dependent on the results of the previous one, it is necessary to terminate the computation after each level and then call the merge task again with the front set to one level further up (see Fig. 2). In other words, coordination are needed and no explicit creation of work units is necessary since the structure is known in advance.

4.2 Collision detection

We use hierarchies to check for collisions between two disjoint objects (inter-object collisions) as well as self collisions for deformable objects (intra-object collisions). The main operation in collision detection is, given several objects, to find whether and at which points they overlap. The objects consist of several geometric primitives (e.g. triangles), which reduces the problem to finding which of these triangles intersect. As a special case, self-intersection tests whether any of the primitives of the object intersect each other, which is of use for applications such as cloth simulation. Since many primitives can be very thin, overlaps can be missed if discrete time steps are used. Therefore, *continuous* collision detection algorithms also test whether objects have collided between two time steps, which is typically a much more expensive operations. The naïve solution to collision detection is to test every combination of primitives, which is not practical due to quadratic

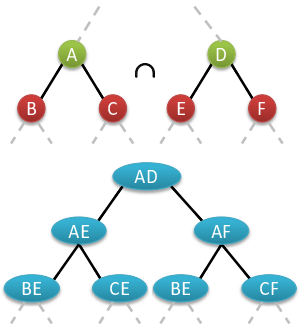

Figure 3: BVTT: *The bounding volume test tree represents the intersections performed during intersection traversal.*

run-time. However, given a hierarchy built on top of each object's primitives, it is possible to intersect the hierarchies instead to find potential collisions and then only perform primitive-primitive intersection for those candidates. This section describes an efficient parallel implementation for this hierarchy intersection using our work organization approach that can be used to perform both discrete and continuous collision detection.

Simultaneous hierarchy traversal: In general, both the geometric primitives as well as the bounding volumes used in the hierarchy can vary depending on the application and other criteria. However, the algorithm for intersecting hierarchies is the same. Starting with the two roots, two hierarchy nodes are intersected by analytically testing their respective bounding volumes for overlap. If they overlap, then all possible pairings of their children are recursively tested for intersection as well. If both of the nodes are leafs, then the two corresponding primitives are put on the list of potential intersections. If only one of the two is a leaf, then it is tested against the children of the other node. This can be seen as traversing a tree of possible bounding volume node pairs, also called the bounding volume test tree [Larsen et al. 1999] (BVTT, see Fig. 3 for illustration.) Similar to the approaches described above, we try to implement this traversal as parallel as possible.

From the standpoint of our approach, the main work units are pairs of hierarchy nodes and for a binary tree each intersection can generate up to four pairs per step. All intersection tests between the node of the hierarchy can be performed independently. As such, the mapping to our work model are relatively simple. The intersection kernel can be run using the vector units to process several intersections in parallel and push the resulting new intersection pairs on the work queue or in a separate result queue for actual primitive pairs. After this traversal, the overall list of primitive pairs is then used as input into a intersection test kernel that tests for actual intersection. Because all potential intersections can be tested fully in parallel, this step is simple to implement by starting enough tasks for all the pairs.

BVTT traversal: One problem with this implementation is that there is a lack of available parallelism while testing the higher levels of the hierarchy that reduces the overall performance of the algorithm. However, it is possible to exploit temporal coherence in the traversal to drastically increase initial available parallelism. During collision detection or cloth simulation, the hierarchy intersection is called for each frame after some deformation has occurred. Considering that the deformation or dynamic movement compared to the last frame may only have been minor, we can use the intersection information as a starting point for the current collision detection [Ehmann and Lin 2001]. To do so, we keep the front in the BVTT

```
INPUT: queue workQueue, TreeNode tree, queue bvttFront
while !workQueue.isEmpty() do
  nodePair ← workQueue.pop()
  leftNode ← tree[nodePair.left]
  rightNode ← tree[nodePair.right]
  if intersects(leftNode,rightNode) then
    {...normal traversal algorithm...}
  else
    parentPair ← (leftNode.parent, rightNode.parent)
    leftNode ← tree[parentPair.left]
    rightNode ← tree[parentPair.right]
    if intersects(leftNode,rightNode) then
      {parent nodes still intersect so at least one sibling must}
      bvttFront.push(parentPair)
    else if parentPair.isLeftMostSibling() then
      {only leftmost sibling creates work unit for parent pair}
      workQueue.push(parentPair)
    end if
  end if
end while
```

Figure 4: BVTT front traversal: *Pseudo-code showing how to traverse using the BVTT front from the last frame. Please see the text for details.*

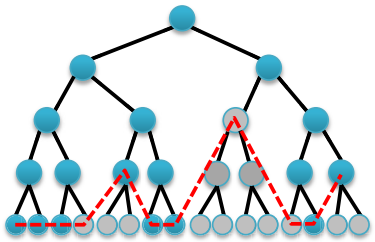

Figure 5: BVTT front: *During the intersection operation, only some BVTT nodes are explored (in blue), which others (grey) never are since their ancestors did not intersect. By tracking the front of blue nodes in the (implicit) hierarchy from the last intersection, we can use it as a starting point for intersection for the next test of the hierarchies.*

(please see Fig. 5 for illustration and [Ehmann and Lin 2001] for more information) which consists of all the intersecting leaf node pairs as well as every non-intersecting node pair for which a sibling intersects. This simple list of node pairs can be written out during our normal traversal. For the next frame, we use this list as input for the work balancing kernel and use the same pairs as the starting work units. The traversal kernel for processing this front is a modified version of the standard traversal algorithm: for each initial node, the intersection is tested again. If the pair intersected last frame and intersects again, then the triangle pair is written to the intersection queue. If the pair did not intersect last frame, but does now, then new work units are created as in the normal traversal. Finally, if the pair does not intersect, then the kernel loads the parents of both nodes and tests them for intersection. If they still intersect, then we have at least one sibling that must still overlap and the node pair is kept in the front. Otherwise, the front is moved upwards by adding the pair of parent nodes to the work queue for the next step. However, since all the siblings are in the front by definition, the parent pair would be added to the list multiple times, which is to be avoided. Instead, we check whether the pair is the leftmost child in the BVTT and only add the parent pair if that is the case to avoid duplication. Fig. 4.2 shows this in a pseudocode implementation.

Model	Tris	Build	AABB	Build	OBB	Refitting		CPU refit
		Compaction	Balancing	Compaction	Balancing	AABB	OBB	AABB
Flamenco	49k	23	22	28	27	1.73	4.6	14
Princess	40k	22	20	26	24	1.4	3.9	12
Sphere/Cloth	92k	35	31	43	39	2.5	7.9	29
Balls	146k	62	56	74	68	3.2	11.5	56

Figure 7: Hierarchy construction and refitting: *Timings of our approach (in ms) for constructing a hierarchy from scratch as well as refitting it, using either AABBs or OBBs. We compare our timings for construction with a compaction-based work approach such as proposed in previous work, and the refitting times to a multi-core CPU version that statically allocates a sub-tree to each CPU (on 4 cores).*

Model	Discrete		Continuous	
	AABB	OBB	AABB	OBB
Flamenco	35	29	40	38
Princess	22	28	27	34
Sphere/Cloth	33	49	47	43
Balls	85	75	99	81

Model	Discrete		Continuous	
	AABB	OBB	AABB	OBB
Flamenco	27	22	37	34
Princess	17	22	26	29
Sphere/Cloth	28	36	42	38
Balls	78	70	91	74

Figure 8: Performance results: *Results for our collision detection implementation. Top: standard traversal. Bottom: front-based traversal. All numbers in ms and including refitting, traversal and triangle intersection.*

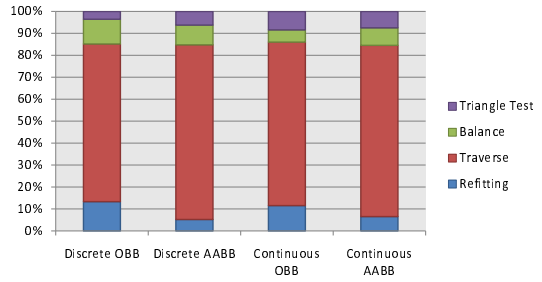

Figure 9: Split-up of timings: *The fraction of time spent in the parts of the algorithm differ based on whether continuous or discrete collision detection performed and which bounding volume is used. In general, for OBBs more time is spent refitting and traversing, but less in intersection.*

5 Implementation

We have implemented our approach using a Intel Core2 Duo system at 2.83 GHz on 4 cores. We are using CUDA on a NVIDIA GTX 285 GPU that has a total of 30 processing cores and 1 GB of memory. Our collision detection algorithm uses a variant of the Moeller test [Moller 1997] for discrete triangle-triangle intersection. For continuous triangles, we solve the cubic equation [Provot 1997] for each of the 15 elementary vertex/face and edge/edge tests to compute intersection status and time of collision. The OBB-OBB intersection test is implemented with the separating axis method [Gottschalk et al. 1996]. Using our work balancing approach, we set the criterion for performing a balancing step to half the tasks being idle. Note that since we launch more tasks than cores to allow hardware multi-threading, this does not result in half the processing cores being idle.

6 Results and Analysis

6.1 Results

We now demonstrate results from our implementation of the algorithms described in the previous sections. We use several common benchmark scenes from previous work to allow easier comparison with other techniques (see Fig. 6). These scenes range from 40k to 146k triangles each and have relatively complex structure. For example, the Flamenco model has several cloth layers very close together, representing a hard case for culling in collision detection.

Fig. 7 shows our results for both construction of our BVH as well as refitting the BVH to dynamic geometry. We use both axis-aligned bounding boxes (AABBs) as well as more complex and tighter-fitting oriented bounding boxes (OBBs) for both approaches. For OBB construction, similar to previous approaches we perform splitting with axis-aligned planes first and then fit the OBB around the triangles in a post-processing step similar to refitting using the PCA approach from [Gottschalk et al. 1996]. We compare the impact of our approach for hierarchy construction compared to previous compaction-based implementations [Zhou et al. 2008; Lauterbach et al. 2009], but improvements are relatively minor at about 10-13% faster timings. We also implemented a simple multi-threaded AABB update running on 4 cores that decomposes the hierarchy into several roughly equal-size subtrees, updates them with parallel threads and merges the results. We found that this implementation is memory limited and provides limited scaling beyond 2 cores and thus is about an order of magnitude slower than our GPU based approach.

Collision detection performance is summarized in Fig. 8. We provide results both for discrete as well as continuous versions using either AABB or OBB bounding volumes. In addition, we show the impact of exploiting temporal coherence by using our front-based traversal implementation of the same algorithms. All numbers are averaged over the whole animation. Note that AABBs are slightly faster on most benchmarks for discrete collision detection where intersections are relatively cheap. However, for continuous collisions where better culling performance is more important, OBBs provide better performance despite their higher cost for maintenance and traversal operations. The front-based traversal generally results in a speedup compared to full traversal, although the result is model-dependent since temporal coherence can vary. The overall impact of any front caching method is limited since by definition at the very minimum all the leaf in the BVTT need to be tested, which will be half the total tested nodes in any case. Thus, even for an unchanged frame the theoretical maximum speedup would be half the traversal time.

We also look at a breakup of timings to show where time is spent in collision detection, i.e. refitting, traversing, balancing and triangle intersection (see Fig. 9), at the example for the Flamenco model. The results show that a large part of the time is spent in the traversal part while intersection makes up a smaller percentage, mostly due to the fact that it runs with optimal parallel utilization given that all intersections can be run independently. Refitting takes a relatively constant fraction of time.

6.2 Analysis

The performance results also show interesting implications on the choice of bounding volume for collision detection. Most current CPU approaches use AABBs since they provide very fast intersection and update operations and are relatively compact in memory.

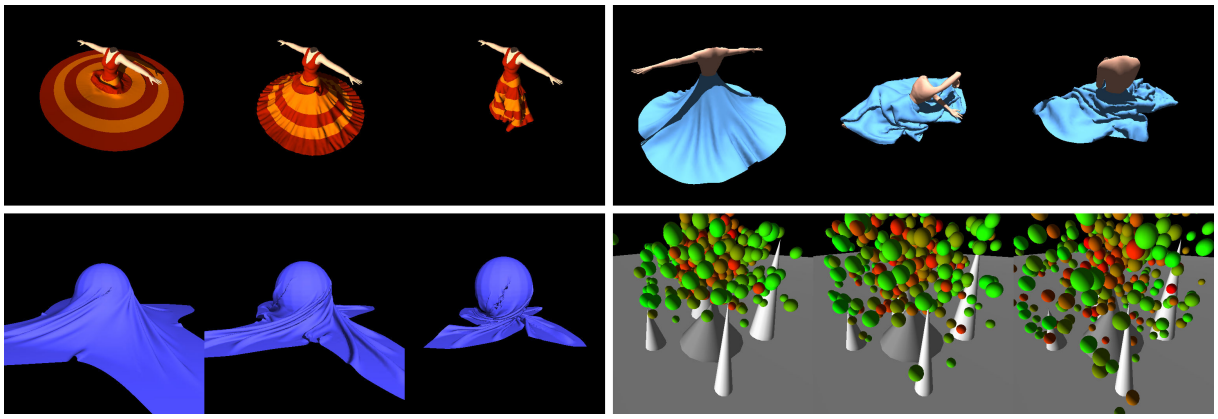

Figure 6: Benchmarks: *The benchmark scenes used in this paper. Top left: Flamenco (49K triangles), top right: Princess (40K), bottom left: Cloth dropping on sphere (92K) and bottom right: n-body simulation (146K). Our algorithm can perform interactive self-collision using continuous triangle intersection on all of these models.*

Our results show that for discrete collision detection, that is the case even in our system. However, for continuous collision detection this situation is reversed and OBBs provide faster results, mainly due to their far better culling efficiency that results in less costly triangle-triangle intersections to be processed. We have found that OBB intersection and refitting benefits from providing much higher *compute density* that is a good match to the high computational power of GPUs. In particular, OBBs use 2.5 times the memory of AABBs, but operations such as computing an OBB from triangles or intersecting two OBBs take about two orders of magnitude as many instructions. For example, AABB-AABB intersection needs just 6 operations but needs to load 12 coordinates to do so. In contrast, OBB-OBB intersection loads 30 coordinates, but then needs to compute 15 separating axis tests, resulting in hundreds of operations. This will be slower on a low-latency CPU architecture with less compute power, but fast on a GPU architecture where high-latency memory accesses are expensive and computational power is high.

Our approach has some limitations worth noting. For one, getting high performance for collision detection depends on having a relatively large front in the BVTT. Unlike self-collision, collision between two different hierarchies, e.g. of deformable models, may exhibit a much smaller front unless the objects are very close such that there are many bounding volume overlaps. In general, even though our approach tries to implement very lightweight synchronization, we are still limited by memory latency for communication. To achieve better scaling, our approach could best benefit from the introduction of a low-latency communication channel between cores on GPU architectures. This would allow the implementation of algorithms such as work stealing during kernel execution, which based on our experiments can already be implemented on current architectures, but are very slow.

6.3 Comparison

Our results on hierarchy construction suggest that the improvement achievable using our approach is relatively minor, which is probably due to the relatively few queue elements (compared to collision detection) and relatively heavy-weight computation kernels which mitigate the impact of the more inefficient compaction algorithms [Zhou et al. 2008; Lauterbach et al. 2009]. However, the fact that hierarchy refitting for both AABBs and OBBs is about an order of magnitude faster suggests that, similar to CPU approaches, updating instead of rebuilding the hierarchy could significantly improve performance in applications such as ray tracing.

Several previous approaches have been proposed to perform self-collision on GPUs using rasterization algorithms [Heidelberger et al. 2003; Knott and Pai 2003; Govindaraju et al. 2003]. Govindaraju *et al.* [2005] used occlusion queries to detect continuous triangle self-collisions for cloth simulation and were able to test a 13K triangle version of the 40K Princess model at about 500ms. Although it is hard to compare performance across GPU generations, occlusion query performance has not scaled with overall computational power. Our approach tests the same model at three times the complexity thirty times faster. Similarly, Sud *et al* [2006] performs self-collision by utilizing discrete Voronoi diagrams generated by rasterization. Their approach took 800ms on a 15K version of the Cloth/Ball scene we are using whereas our approach is over 25x faster on the full 92K model.

The fastest CPU approaches for continuous collision detection have employed feature-based hierarchies and other culling methods to reduce the amount of intersection tests to be performed, such as in [Curtis et al. 2008]. Their implementation on a single-core was able to test for self-collision in 200ms per frame on the Flamenco model by improving the performance several times over a standard simulation. These culling approaches are orthogonal to our work and could be integrated into our framework as well. More recent work has investigated parallelization methods on multi-core CPU systems. The implementation presented by Kim *et al.* [2008] achieved speedups from 6.4x to 7.3x on an 8 core system. Comparing on the same benchmarks, we note that our GPU implementation provides similar or better performance for continuous collision detection.

7 Conclusion and future work

We have presented a new, lightweight work balancing algorithm for GPU hierarchy algorithms and showed its use in applications such as hierarchy construction and refitting as well as discrete and continuous collision detection. We have also introduced a parallel front-based traversal method for collision detection that greatly reduces the bottlenecks in collision traversal. Our results show that we can compute and use tight-fitting bounding volumes interactively and perform continuous collision detection much faster than previous approaches and competitively or faster than current multi-core CPU methods.

There are many avenues for future work, especially in other applications. Algorithms that use hierarchies include n-body simulation, point-based algorithms and many more. Enabling their efficient implementation on GPUs is important for high-performance

implementations. Other applications similar to collision detections such as proximity queries are also widely used and could be implemented similarly. For self-collision, many techniques have been proposed to reduce the amount of edge/edge and vertex/face intersection test significantly [Curtis et al. 2008]. Our approach is mostly orthogonal, so it would be interesting to investigate implementing these methods in our framework.

References

ARORA, N. S., BLUMOFE, R. D., AND PLAXTON, C. G. 1998. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, ACM, New York, NY, USA, 119–129.

CHASE, D., AND LEV, D. 2005. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, ACM, New York, NY, USA, 21–28.

CUNTZ, N., AND KOLB, A. 2007. Fast Hierarchical 3D Distance Transforms on the GPU. 93–96.

CURTIS, S., TAMSTORF, R., AND MANOCHA, D. 2008. Fast collision detection for deformable models using representative-triangles. *Proc. of ACM Symposium on Interactive 3D Graphics and Games*.

EHMANN, S., AND LIN, M. 2001. Accurate and fast proximity queries between polyhedra using surface decomposition. In *Proc. of Eurographics*.

ERICSON, C. 2004. *Real-Time Collision Detection*. Morgan Kaufmann.

FOLEY, T., AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proc. ACM SIGGRAPH/EG Conf. on Graphics Hardware*, 15–22.

FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. 1998. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not. 33*, 5, 212–223.

GORADIA, R., AJMERA, P., AND CHANDRAN, S. 2008. Gpu-based hierarchical computations for view independent visibility. In *ICVGIP*, 560–567.

GOTTSCHALK, S., LIN, M., AND MANOCHA, D. 1996. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, 171–180.

GOVINDARAJU, N., REDON, S., LIN, M., AND MANOCHA, D. 2003. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 25–32.

GOVINDARAJU, N., KNOTT, D., JAIN, N., KABAL, I., TAMSTORF, R., GAYLE, R., LIN, M., AND MANOCHA, D. 2005. Collision detection between deformable models using chromatic decomposition. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH) 24*, 3, 991–999.

GRINBERG, I., AND WISEMAN, Y. 2007. Scalable parallel collision detection simulation. *Proc. of Signal and Image Processing*.

GUTHE, M., BALÁZS, A., AND KLEIN, R. 2005. Gpu-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Trans. Graph. 24*, 3, 1016–1023.

HEIDELBERGER, B., TESCHNER, M., AND GROSS, M. 2003. Real-time volumetric intersections of deforming objects. *Proc. of Vision, Modeling and Visualization*, 461–468.

HENDLER, D., AND SHAVIT, N. 2002. Non-blocking steal-half work queues. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, ACM, New York, NY, USA, 280–289.

HENDLER, D., LEV, Y., MOIR, M., AND SHAVIT, N. 2006. A dynamic-sized nonblocking work stealing deque. *Distrib. Comput. 18*, 3, 189–207.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree GPU raytracing. In *Proc. I3D '07*, 167–174.

KIM, D., HEO, J.-P., AND EUI YOON, S. 2008. Pccd: Parallel continuous collision detection. Tech. rep., Dept. of CS, KAIST, Technical Report CS-TR-2008-298.

KITAMURA, Y., SMITH, A., TAKEMURA, H., AND KISHINO, F. 1995. Parallel algorithms for real-time colliding face detection. *Robot and Human Communication, 1995. RO-MAN'95 TOKYO, Proceedings., 4th IEEE International Workshop on* (Jul), 211–218.

KNOTT, D., AND PAI, D. K. 2003. CInDeR: Collision and interference detection in real-time using graphics hardware. *Proc. of Graphics Interface*, 73–80.

KUMAR, V., AND GRAMA, A. Y. 1994. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing 22*, 60–79.

KYLE HEGEMAN, N. A. C., AND MILLER, G. S. 2006. Particle-based fluid simulation on the gpu. 228–235.

LARSEN, E., GOTTSCHALK, S., LIN, M., AND MANOCHA, D. 1999. Fast proximity queries with swept sphere volumes. Tech. Rep. TR99-018, Department of Computer Science, University of North Carolina.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast bvh construction on gpus. In *Proc. Eurographics '09*.

LE GRAND, S. 2007. Broad-phase collision detection with cuda.

LEFEBVRE, S., AND HOPPE, H. 2006. Perfect spatial hashing. In *SIGGRAPH '06*, ACM, New York, NY, USA, 579–588.

LEFEBVRE, S., AND HOPPE, H. 2007. Compressed random-access trees for spatially coherent data. In *Rendering Techniques (Proceedings of the Eurographics Symposium on Rendering)*, Eurographics.

LEFOHN, A., KNISS, J. M., STRZODKA, R., SENGUPTA, S., AND OWENS, J. D. 2006. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics 25*, 1 (Jan.), 60–99.

LEFOHN, A. E., SENGUPTA, S., AND OWENS, J. D. 2007. Resolution-matched shadow maps. *ACM Trans. Graph. 26*, 4, 20.

MOLLER, T. 1997. A fast triangle-triangle intersection test. *Journal of Graphics Tools 2*, 2.

PROVOT, X. 1997. Collision and self-collision handling in cloth model dedicated to design garment. *Graphics Interface*, 177–189.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 703–712.

PURCELL, T., DONNER, C., CAMMARANO, M., JENSEN, H., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, 41–50.

RAO, V. N., AND KUMAR, V. 1987. Parallel depth-first search, part i: Implementation. *International Journal of Parallel Programming 16*, 6–479.

SUD, A., GOVINDARAJU, N., GAYLE, R., KABUL, I., AND MANOCHA, D. 2006. Fast proximity computation among deformable models using discrete voronoi diagrams. *Proc. of ACM SIGGRAPH*, 1144–1153.

SZIRMAY-KALOS, L., SZÉCSI, L., AND SBERT, M. 2008. *GPU-Based Techniques for Global Illumination Effects*. Morgan & Claypool.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. In *Proc. SIGGRAPH Asia*.