Collision-Streams: Fast GPU-based Collision Detection for Deformable Models

Min Tang[1] Dinesh Manocha[2] Jiang Lin[1] Ruofeng Tong[1]
[1]Zhejiang University [2]University of North Carolina at Chapel Hill
http://gamma.cs.unc.edu/CSTREAMS/

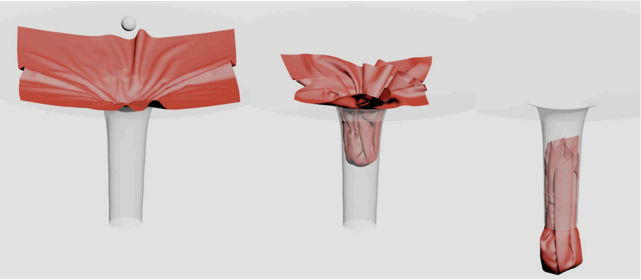

Figure 1: Benchmark Funnel: *In this simulation, a cloth falls into a funnel and pass through it under the pressure of a ball. This model has $47K$ vertices, $92K$ triangles, and a lot of self-collisions. Our novel GPU-based CCD algorithm takes $4.4ms$ and $10ms$ per frame to compute all the collisions on a NVIDIA GeForce GTX 480 and a NVIDIA GeForce GTX 285, respectively.*

Abstract

We present a fast GPU-based streaming algorithm to perform collision queries between deformable models. Our approach is based on hierarchical culling and reduces the computation to generating different streams. We present a novel stream registration method to compact the streams and efficiently compute the potentially colliding pairs of primitives. We also use a deferred front tracking method to lower the memory overhead. The overall algorithm has been implemented on different GPUs and we have evaluated its performance on non-rigid and deformable simulations. We highlight our speedups over prior CPU-based and GPU-based algorithms. In practice, our algorithm can perform inter-object and intra-object computations on models composed of hundreds of thousands of triangles in tens of milliseconds.

1 Introduction

The problem of checking whether the primitives of two or more objects overlap with each other arises in different applications including physics-based simulation, robot motion planning, haptic rendering and virtual environments. The underlying simulator needs to check for collisions between a pair of objects (inter-object collisions) as well as self-collisions among deformable objects (intra-object collisions). Recent work in this area has been on continuous collision detection (CCD), which checks for collisions between two discrete time instances by interpolating a smooth motion. Moreover, many interactive applications must perform this computation at $30 - 60$Hz or higher rates.

There is extensive work on designing fast algorithms for collision detection based on bounding volume hierarchies (BVHs). Recent work includes development of appropriate culling techniques that tend to reduce the number of false positives in terms of pairwise intersection tests between the primitives. Most of the earlier algorithms were serial in nature and designed for a single core or processor. However, the recent trend in computer architecture has been toward developing parallel commodity processors, including multi-core CPUs and many-core GPUs. It is expected that the number of cores would increase at the rate corresponding to Moore's Law. Given these trends, many parallel collision detection algorithms have been proposed for commodity parallel processors. In this paper, we mainly deal with designing fast GPU-based algorithms that can exploit the data and thread-level parallelism and are flexible in terms of using various culling techniques.

Modern GPUs can be regarded as many-core stream processors which offer higher peak throughput as compared to CPUs. However, the underlying architecture and the memory hierarchy of GPUs imposes some constraints in terms of designing appropriate algorithms. As a result, many prior GPU-based collision detection algorithms exploit rasterization capabilities [Heidelberger et al. 2003; Knott and Pai 2003; Govindaraju et al. 2003; Greß et al. 2006; Sud et al. 2004; Morvan et al. 2008], or perform explicit balancing between the work units [Lauterbach et al. 2010] or use hybrid combination of CPU and GPU to distribute the computation [Govindaraju et al. 2005; Kim et al. 2009; Pabst et al. 2010]. It is relatively hard to combine different culling methods or optimizations along with these GPU-based algorithms. Without using these culling methods, a lot of computation ability and memories will be waste on redundant tests and false positives (up to 2 orders of magnitude [Tang et al. 2009; Tang et al. 2010a]).

Our contributions: We present a novel GPU-based collision detection algorithm that abstracts a GPU as a stream processor that is good at handling stream data in parallel with kernels. Our formulation is based on BVHs and reduces the computation to generating various streams. We also present a novel stream registration method to efficiently support variable-length data structures, which can be updated in parallel by multiple threads. This provides us the flexibility to incorporate many culling methods based on Orphan sets [Tang et al. 2009], Representative triangles [Curtis et al. 2008] and non-penetration filters [Tang et al. 2010a], which can improve the overall performance of the GPU-based algorithm. We also use a deferred front tracking scheme that reduces the memory overhead and makes it possible to handle large complex models on GPUs. The overall algorithm, collision-streams, maps well to current GPU architectures and we evaluate its performance on NVIDIA GeForce GTX GPUs and a AMD Radeon HD 5870 GPU.

The overall collision detection algorithm makes no assumption about the objects or motion, can detect all inter-object and intra-object collisions at object-space precision, and minimizes the number of elementary tests between adjacent triangles. We also analyze the performance of our algorithm as a function of number of cores or streaming processors. In practice, collision-streams can perform CCD computations between complex deformable models composed of hundreds of thousands of triangles in tens of milliseconds. We highlight our speedups over prior CPU-based and GPU-based algorithms and a lower memory overhead. Moreover, it is relative easy to combine other culling techniques.

Organization: The rest of the paper is organized as follows: Section 2 gives a brief survey of prior work. We present our streaming algorithm, along with streaming registration and deferred front tracking in Section 3. We present the implementation details and highlight the performance in Section 4. We compare our approach with prior algorithms in Section 5.

2 Related Work and Background

In this section, we give a brief overview of related work on collision detection and GPU-based algorithms. We also highlight many characteristics of GPU architectures and abstract them as streaming processors.

Continuous Collision Detection: Many techniques have been proposed to accelerate the performance of collision detection algorithms. These include bounding volume hierarchies (BVHs) including AABB trees [van den Bergen 1997], OBB trees [Gottschalk et al. 1996], k-DOP trees [Klosowski et al. 1998], etc. These hierarchies need to be updated for deformable models based on refitting methods or selective restructuring [Larsson and Akenine-Möller 2006; Lauterbach et al. 2006; Zachmann and Weller 2006; Otaduy et al. 2007]. Many techniques have been proposed to improve the performance of CCD algorithms, including feature-based bounding volume [Hutter and Fuhrmann 2007], non-penetration filters [Tang et al. 2010a], continuous normal cones [Tang et al. 2009], etc. Other techniques can significantly reduce the number of elementary tests between the primitives based on Representative triangles [Curtis et al. 2008] or Orphan sets [Tang et al. 2009]. In practice, these methods can be combined with bounding volume hierarchies and can improve the overall performance of CCD algorithms by more than an order of magnitude for CPU-based algorithms. In this paper, our goal is to develop appropriate GPU-based algorithms where such culling techniques can be used to reduce the number of elementary tests.

Collision Detection on Commodity Parallel Processors: In order to exploit the computational capability of commodity processors, the recent trend has been to design parallel collision detection algorithms. This includes multi-core algorithms for current CPUs [Tang et al. 2010b] using front-based decomposition that exploit task-level and data-level parallelism. There is extensive work on exploiting the computational power of commodity GPUs (graphics processing units) for fast collision and proximity computations. Some of the earlier methods exploited the rasterization capabilities of GPUs to perform collision and distance queries [Heidelberger et al. 2003; Knott and Pai 2003; Govindaraju et al. 2003; Greß et al. 2006; Sud et al. 2004; Morvan et al. 2008] at image-space resolution. Many hybrid combinations of GPU and CPU algorithms have been proposed to perform collision and distance queries at object-space resolution [Govindaraju et al. 2005; Sud et al. 2006; Kim et al. 2009]. Other GPU-based methods are designed for broad-phase culling [Grand 2007] and AABB streaming [Zhang and Kim 2007]. Some of the recent work uses GPUs as general-purpose processors to perform fast hierarchy traversal [Lauterbach et al. 2010] or spatial hashing [Pabst et al. 2010]. GPU-based parallel algorithms have also been proposed to construct BVHs [Lauterbach et al. 2009; Pantaleoni and Luebke 2010] and octrees [Zhou et al. 2010].

Stream Computing Using GPUs: Current GPUs are regarded as high-throughput processors, which have a theoretical peak performance of a few Tera-Flops. Most of these GPUs operate on a SIMD (single-instruction multiple data) basis and the computations are performed simultaneously by executing a large number of threads. At a broad level, the GPUs consist of several streaming multi-processors, with each of them containing a number of streaming processors and a small shared memory unit. For example, the latest NVIDIA GeForce GTX 480 GPU has 480 processor cores, and the latest Radeon HD 5870 GPU from AMD has 320 processors and each of those processors have 5 ALUs. Moreover, there is global memory that is accessible to all the streaming multiprocessors. The GPU memory system provides a higher bandwidth as compared to the CPU memory system, but has a higher latency. The caches used in the GPUs are relatively small as compared to CPUs and recent GPUs can support a two-level cache hierarchy.

The GPUs can be abstracted as stream processors that are good at handling stream data in parallel with kernels [Nickolls and Dally 2010]. In this model, the underlying program structure can be described by streams of data passing through computation kernels. A stream is an ordered set of data of an arbitrary (simple or complex) data type and a kernel performs operations on streams or substreams in parallel. In essence, the underlying kernels operate on one or more streams as inputs and produce one or more streams as outputs. Furthermore, the overall program can be constructed by chaining these computations together. This formulation has been used to design efficient GPU-based sorting and numerical computations [Owens et al. 2007]. On current GPUs, the kernels are executed by multiple threads which are organized into a two-level hierarchy: blocks and threads. At the top level of the hierarchy, a grid is organized as a two dimensional array of blocks. At the bottom level, all blocks of a grid are organized into an array of threads. All the threads in the same block can access a small, high-speed shared memory. However, threads from different blocks can only communicate via slower global memory, and can only synchronize via expensive atomic operations.

2.1 Stream Compaction

Stream compaction is often used on GPUs to remove unwanted elements in a sparse data representation. In our streaming CCD algorithm, several streams containing sparse data (e.g. potentially colliding triangle pairs or potentially colliding feature pairs) are used and need to be compacted for good performance. The actual stream data elements used is much less than $O(N^2)$ possible pairs (N is the number of total triangles in the scene), and therefore the resulting data structure needs to support sparse data representations.

Many fast stream compaction [Horn 2005; Harris et al. 2007; Billeter et al. 2009] and scan algorithms have been designed for current GPUs [Sengupta et al. 2011; Merrill and Grimshaw 2009]. In order to use these methods for generating collision streams, the resulting algorithm would require $O(N^2)$ storage, which would limit their application to relatively simple models.

3 Streaming Continuous Collision Detection

In this section, we introduce our notation and present our CCD algorithm based on collision-streams. We assume that the scene consists of multiple deformable objects. Our algorithm builds a BVH for the entire scene and performs top-down traversal to check for both inter-object and intra-object collisions. We make no assumption about the motion or the deformation and assume that we are given the positions of all the vertices of the mesh at each time frame. In terms of CCD computation, the algorithm assumes linearly interpolating motion between the vertices and the first time of contact is computed by performing elementary tests between the features (i.e. vertices, edges and faces).

3.1 Algorithm Overview

Our algorithm is based on BVHs and at a broad level consists of the following stages:

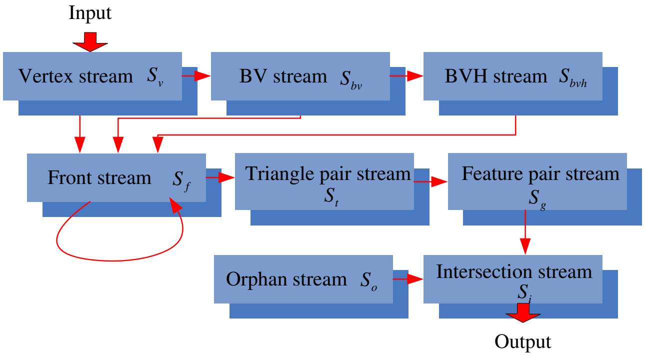

Figure 2: The stream data: *All the geometric data, such as vertex coordinates, BVs, BVH, BVTT front, Orphan set, triangle pairs, feature pairs, and colliding information are represented as GPU streams. Different kernels operate on these streams. The thin arrows refer to dependency relationship between these streams. The current "Front stream" is refined from the stream at previous frame (shown as a self-reference).*

1. At each frame, update the triangle positions, bounding volumes, and the BVH for the whole scene using refitting algorithms.
2. Traverse the BVH and perform pairwise tests between the bounding volumes (BVs) to compute the list of potentially intersecting triangles. This step can be accelerated by maintaining a front of the resulting BVTT (bounding volume traversal tree) [Tang et al. 2010b] and thereby reducing the number of pairwise BV tests.
3. For all potentially colliding triangle pairs, elementary tests are performed between their features (vertices, edges, and triangles). The CCD test between each pair of triangles reduces to 15 elementary tests, including 6 VF (vertex-face) and 9 EE (edge-edge) tests. The number of elementary tests can be reduced based on Orphan sets [Tang et al. 2009] and Representative triangles [Curtis et al. 2008]. Each elementary test reduces to solving a cubic equation, and it can be accelerated by using a non-penetration filter [Tang et al. 2010a].

In order to utilize the computation capabilities of GPUs, we map the hierarchical traversal and intersection tests to collision-streams, which can exploit the parallelism on current GPUs.

3.2 Collision Streams

In order to fully utilize the powerful parallel computation ability of current GPUs, we model them as a collection of stream processors that can perform large-scale fine-grained parallel computation on stream data. In terms of the overall CCD algorithm, the geometric data are represented as stream data, and the underlying functional modules used in the algorithm (i.e. updating BVHs, pairwise tests between BVs, elementary tests, etc.) are mapped to computation kernels.

The stream data corresponding to the geometric data includes (Figure 2):

- **Vertex stream** S_v: These correspond to the geometric coordinates of the vertices of the deformable objects. In order to perform continuous collision detection between two discrete time steps, two vertex streams $S_v(t_0)$ and $S_v(t_1)$ are used to store the vertex coordinates corresponding to time t_0 and t_1, respectively.

Algorithm 1 Collision-stream algorithm for CCD Computation.

```
Construct orphan stream $S_o$, initial BVH $B$ of the scene, and initial BVTT front stream $S_f$ on a CPU
Send $S_o$, $B$, $S_f$ from CPU to GPU
for each simulation time step perform the following computations on the GPU do
    Update vertex stream $S_v$ from current vertices
    Update BVs and BVH ($S_{bv}$ and $S_{bvh}$) with $S_v$
    Update the front $S_f$, and generate the triangle pair stream $S_t$
    Generate feature pair stream $S_g$ from $S_t$
    Perform exact tests on $S_g$ and collect result into $S_i$
end for
```

- **BV stream** S_{bv} and **BVH stream** S_{bvh}: All the bounding volumes for triangles, edges, and vertices are represented by BV stream S_{bv}. The whole scene is enclosed by a single BVH S_{bvh}. S_{bv} and S_{bvh} are updated at each simulation time step based on S_v.
- **Front stream** S_f: These are the nodes corresponding to the BVTT front, i.e., the internal nodes and leaf nodes where the tree traversal terminated in the previous time step.
- **Triangle pair stream** S_t: During BVH traversal, all the non-adjacent triangle pairs whose bounding volume overlap are collected into S_t. These potentially colliding triangle pairs are represented in terms of different features, including vertices, edges and faces, before performing elementary tests. We use Representative triangles [Curtis et al. 2008] to ensure a feature shared by multiple triangles is not replicated. This reduces the storage overhead and number of elementary tests.
- **Feature pair stream** S_g and **Orphan stream** S_o: All the non-adjacent triangle pairs with overlapping bounding volumes are decomposed into at most 9 EE and 6 VF feature pairs. These feature pairs are further culled using BVs associated with the features and deforming non-penetration filters [Tang et al. 2010a]. All the pairs that satisfy these tests become part of a feature-pair stream S_g. Moreover, all invalid feature pairs along with culled feature pairs are removed by performing stream compaction on S_g. All the feature pairs related to adjacent triangles in the mesh are represented as part of the Orphan stream S_o. S_o is constructed at the pre-processing stage by analyzing the topology and connectivity of the mesh corresponding to each object.
- **Intersection Stream** S_i: All the feature pairs in the feature pair stream S_g and the Orphan stream S_o are tested for overlap between t_0 and t_1 by solving a cubic equation. If a collision is found, the feature IDs and first-time-of-contact information is stored in the Intersection stream S_i.

The pipeline of our algorithm is shown in Algorithm 1. All the procedures are mapped to a set of computation kernels (as shown in Figure 3). By executing these kernels, the streaming CCD algorithm runs entirely on a GPU.

3.3 Updating Front Stream

In order to perform the collision detection tasks in parallel, our algorithm uses a task-decomposition strategy based on parallel updating of the BVTT front, i.e., by utilizing the temporal coherence between successive time steps of the deformable simulation. [Tang et al. 2010b].

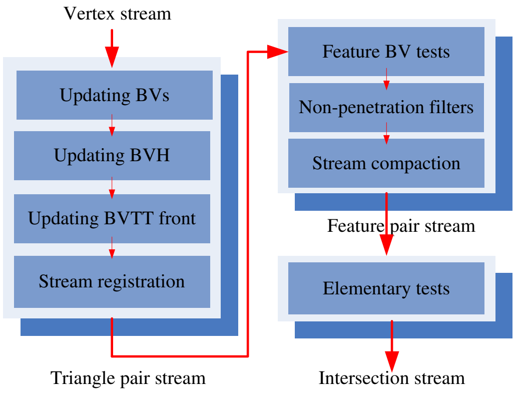

Figure 3: Computation kernels: *The modules, such as updating BVs, updating BVH, regenerate BVTT front, non-penetration filter, and elementary tests, etc., are represented as computation kernels. The kernels, stream registration and stream compaction, are applied to maintain compact stream data.*

The BVTT front from the previous simulation time step is represented as a front stream $S_f(t_0)$. It is evaluated by the kernel 'Updating BVTT front' that updates the BVTT front at the current simulation time step, i.e. compute a new front stream $S_f(t_1)$. We use a temporary stack to store intermediate results while perform recursively transverse on a BVTT front node $\{D_a, D_b\}$. The node pairs with overlapping bounding volumes are stored as $S_f(t_1)$. If both D_a and D_b are leaf nodes, this pair is stored in a triangle pair stream S_t, as it corresponds to a potentially colliding pair. The temporary stack is implemented by using register memory on GPU.

Our algorithm is based on front tracking, so only the BVH nodes stored in the BVTT front need to be updated. We used refitting scheme to update all these nodes fully in parallel by using the kernel 'Updating BVH'.

3.4 Stream Registration

For a scene containing deformable objects, the potentially colliding triangles may not have a fixed or coherent pattern and can correspond to arbitrary triangles in terms of their memory location, especially when the object undergoes large deformation. In order to enumerate all these possible pairs of overlapping triangles, a large block of GPU memory of size $O(N^2)$ would be needed. The large memory overhead needed to represent all potential pairs of overlapping triangles or features prevents us from directly using prior stream compaction algorithms. A naive solution of this problem is to use atomic operations (e.g., $atomicAdd()$ in CUDA or $atom\_add()$ in OpenCL) to maintain a compact stream data. Based on these atomic operations, multiple threads of a kernel can mutually access an index variable of an array, and put generated stream data tightly in GPU memory of O(M), here M is an output sensitive metric in terms of number of overlapping primitives. However, these atomic operations can be quite expensive (take hundreds of clock cycles[Supercomputing Blog 2009]) since they tend to suspend the concurrent threads and thereby reduce the overall throughput. Another approach is to use prefix sum, i.e., use a large array to store all possible outcomes of collision detection, then the array is compressed and remove pairwise combinations that do not correspond to a collision pair. However, because the space requirements correspond to the total number of possible pairs $O(N^2)$, this approach may be limited to simple scenes. In order to overcome this issue, a 2-pass prefix sum strategy can be used. During the first pass, the resulting algorithm records how many output elements will be generated, without actually generating any output. During the second pass, it allocates an output array based on the size information computed during the first pass and performs the computations to generate the actual output. However, this 2-pass scheme has additional overhead. In order to overcome these issues, we present *stream registration*, which is a lock-free variable-length data structure that can reduce the memory overhead on GPUs.

Algorithm 2 Stream registration based on locking-free mechanism.

```
// Launch kernel-1:
for each thread $thd_i$ operating on a set of front nodes $set_i$ do
    Record the new nodes into memory segment $seg_i$,
    and update its private index $idx_i$.
end for
// Launch kernel-2:
Merge all memory segments $\{seg_i, idx_i\}$ into
the destination GPU memory with Prefix Sum operator.
```

A primary stream registration algorithm is based on segmented locking mechanism (Figure 4). As shown in the figure, all the threads belonging to the same block perform atomic operations to access an index value of a private array for each block to store new nodes into it. Next all these private arrays are compacted into continuous space by using Prefix Sum operator. Let the number of blocks be M_b , and the number of concurrent threads in each block be M_t. Based on segment locking mechanism, the mutual access of a single index variable by $M_b * M_t$ threads is replaced by M_b independent mutual access of indices by M_t threads. By reducing the coupling degree among the threads, the performance of parallel execution is improved in terms of data access. These private arrays are much smaller than $O(N^2)$. In our implementation, we used a conservative size of the array $(\min(0.025\% \times N^2, 1.4M))$ as the total length of these private arrays which appears to be adequate for our benchmarks.

This approach can be further extended to avoid using atomic operators altogether. The algorithm consists of two kernels. As shown in Figure 5, we can allocate a private array to each thread to temporarily store the newly generated node, which may correspond to the front nodes in the BVTT or potentially colliding triangle pairs. All the threads running the first kernel can be executed in parallel. Since there is no data sharing between the threads, all the computations in each thread are performed independently, and no atomic operators are used. It turns out if the number of new generated nodes exceeds the size of temporary storage space, atomic operators are used to move these new nodes to the global GPU memory. After executing the first kernel, the new nodes are distributed in memory segments. During the execution of the second kernel, a Prefix Sum operator is used to store these nodes into compact space. Since we assume that the start positions and length of the new nodes in each memory segment are already known, we only need to compute Prefix Sum of each memory segment to decide their new locations in the destination GPU memory and move the new nodes in parallel (Algorithm 2).

3.5 Generating Feature Pair Stream

All the potentially colliding triangle pairs in the triangle pair stream are decomposed into at most 15 feature pairs (9 EE pairs and 6 VF pairs). These feature pairs are culled with feature-based bounding volumes and non-penetration filters. To ensure the compactness of the feature pair stream, those culled feature pairs are removed with Prefix Sum operator [Harris et al. 2007] . In practice, we observe that the length of the feature pair stream can be compressed to 4.2%-9.8% of its original size in our benchmarks.

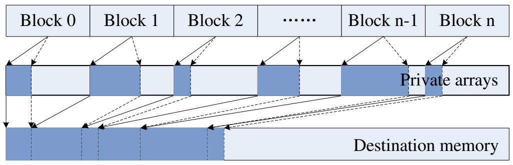

Figure 4: Stream registration based on segmented locking mechanism: *A segmented locking mechanism is used to reduce the overhead of frequent atomic operations. All the threads belonging to the same block use atomic operators to access an index value of a private array of the block to store new nodes into a memory segment. All these private arrays are compacted (as shown in the bottom row) into a continuous space by using Prefix Sum operators. The dark blue areas are collected in each private array and the light blue areas correspond to the unused memory space.*

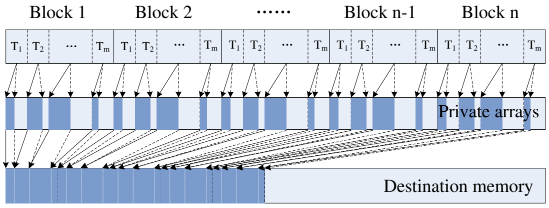

Figure 5: Stream registration based on locking-free mechanism: *By allocating a segment of space of each thread to temporarily store the newly generated nodes, all the threads can be executed in parallel. Since there is no data sharing between threads, all the computations in each thread are performed in an independent manner and no atomic operators are used.*

3.6 Time-of-Contact Computation

All the feature pairs (EE pairs and VF pairs) are tested for overlap by solving cubic equations that can also compute the first-time-of-contact. Newton iterative method is used to solve these cubic equations on a GPU. If there is an overlap between the primitives, that information is recorded in the Intersection stream.

3.7 Deferred Front Tracking

In order to apply our algorithm on complex models, we use a deferred front tracking scheme to reduce the memory overhead. As shown in the left-upper corner of Figure 6, a BVH is constructed for all the objects in the scene. In order to detect both inter-object and intra-object collisions, we use BVTTs to represent the top down traversal. A conventional front tracking algorithm stores all the internal nodes and leaf nodes where the traversal terminates. As part of the collision query, we traverse all the nodes starting from the deferred front, until the nodes of the new exact front is calculated. This involves performing additional overlap tests between the bounding volumes. Overall, we trade off memory overhead for additional runtime computations.

A deferred front (e.g. shown as the red thick curve in Figure 6) stores ancestral nodes of the exact front (the blue thick curve in Figure 6) where the distance to the ancestor is chosen based on memory considerations. From a given ancestor, it is possible to recalculate the exact front by descending the tree on-demand, effectively reducing the memory footprint of the front in exchange for increased computation. The main benefits of this approach are: (1) The memory capacity of commodity GPUs is quite limited. Storing the exact BVTT front could exceed the capability of GPUs for complex models; (2) In order to obtain higher performance on GPUs,

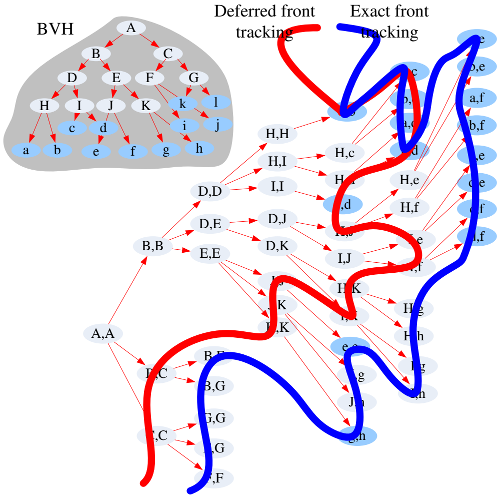

Figure 6: Deferred front tracking vs. exact front tracking: *The conventional front tracking algorithm stores all the internal nodes and leaf nodes where the traversal terminates in the BVTT(the blue thick curve). A deferred front (the red thick curve) stores the ancestors of the nodes belonging to the exact front. By doing so, we can reduce the memory requirement up to 8.8X for large, complex models and enable CCD computation on large scene (up to $2M$ triangles) on commodity GPUs.*

it is recommended to use computation intensive tasks to hide data access latency. Frequently updating the exact front of the BVTT can be an expensive operation, even with the use of stream registration. With deferred front tracking, we can either keep an unchanged front from the last frame or perform relative small update between successive frames.

We base the decision of which ancestor of the actual front will be stored in the deferred front on the memory size of the GPU. We defer only as far up the tree as is needed to make the deferred front fit in memory. In practice, we select the nodes of deferred front depending on the size of the exact front and the capacity of GPU memory. If the exact front is large (for a complex scene), higher level nodes of the BVTT are selected to reduce the memory overhead.

In our benchmarks, we observe 21% speedups for the Cloth benchmark and can reduce the memory overhead by 8.2X. For complex scenes, e.g., deferred font tracking must be used to perform CCD with limited GPU memory. For the Lion benchmark (Figure 8(f)), which consists of 1.6M triangles, over 3GB of memory may be needed to store the exact BVTT front (about 200M nodes of the exact front), while our deferred front tracking only uses 340M (about 2.2M front nodes).

4 Implementation and Performance

In this section, we describe our implementation and highlight the performance of our algorithm on several benchmarks.

4.1 Implementation

We have implemented our algorithm on three different commodity GPUs: a NVIDIA GeForce GTX 285, a NVIDIA GeForce GTX 480, and a AMD Raedon HD 5870. Their parameters are shown in Figure 7. For the two NVIDIA GPUs, we used CUDA toolkit 3.2 as the development environment. For the AMD GPU, OpenCL is used

GPU	GeForce GTX 285	GeForce GTX 480	Raedon HD 5870
Number of Cores	320	480	320
Memory capacity (G)	1.0	1.5	1.0
Memory clock rate (MHz)	1242	1858	1200
GPU clock rate (MHz)	648	700	850

Figure 7: GPUs: *Three different commodity GPUs, a NVIDIA GeForce GTX 285, a NVIDIA GeForce GTX 480, and a AMD Raedon HD 5870, are used for testing our CCD algorithm.*

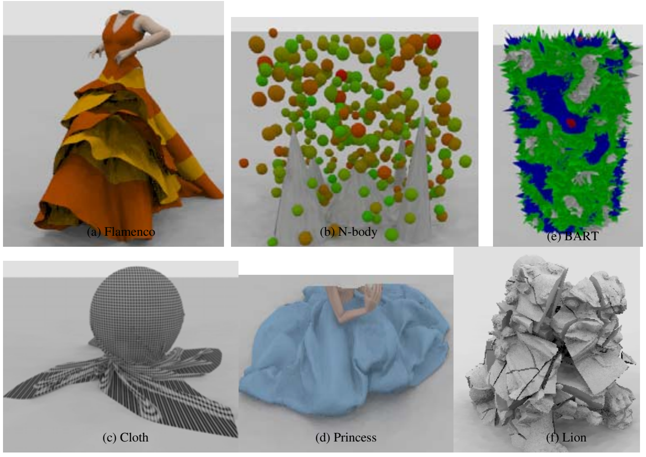

Figure 8: Benchmarks: *All the benchmarks have multiple simulation steps. We perform CCD, including self-collisions, between discrete steps of the simulation and compute the first time-of-contact.*

as the programming environment. We use k-DOPs (specifically 18-DOPs) as the bounding volumes. As compared to AABB, k-DOPs provide higher culling rates and can be dynamically updated at low overhead. The BVH is constructed in a top-down manner with median plane cutting along the longest axis. We use refitting methods to update the hierarchy for deformable models.

4.2 Benchmarks

In order to test the performance of our algorithm, we used six different benchmarks, arising from different simulations with different characteristics.

- **Funnel:** A cloth ($92K$ triangles) falls into a funnel and folds to fit into the funnel with many self-collisions (Figure 1).
- **Flamenco:** This benchmark ($49K$ triangles) has many inter- and intra-object collisions (Figure 8(a)).
- **N-body:** A scene with hundreds of balls ($146K$ triangles) that are colliding with each other (Figure 8(b)).
- **Cloth:** A cloth ($92K$ triangles) has a high number of self-collisions (Figure 8(c)).
- **Princess:** This model ($40K$ triangles) has many inter- and intra-object collisions (Figure 8(d)).
- **BART:** A set of triangles ($4K$ triangles) move in random directory, collide with each other and generate many inter-object collisions (Figure 8(e)).

Benchmarks	GTX 285	GTX 480	HD 5870
Cloth-ball (*92K*)	42.6	18.6	40.5
Funnel (*92K*)	10.0	4.4	4.9
N-body (*146K*)	155.6	79.0	85.3
BART (*4K*)	26.7	10.2	32.4
Flamenco (*49K*)	68.7	32.7	33.6
Princess (*40K*)	7.1	2.7	6.4
Lion (*1.6M*)	1136.5	316.6	432.9

Figure 9: Performance results: *This figure shows the average query time (ms) of our algorithm on a NVIDIA GeForce GTX 285, a NVIDIA GeForce GTX 480, and a AMD Raedon HD 5870, respectively.*

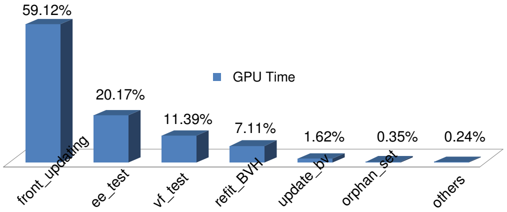

Figure 10: Running time ratios of each kernel : *The figure shows the running time ratios of each kernel for the Funnel benchmark. Updating of BVTT front, elementary tests (EE tests and VF tests), and BVH refitting occupy* 59.12%, 31.56%, *and* 7.11% *of total running time, respectively.*

- **Lion:** A lion ($1.6M$ triangles) is gradually collapsing with topology changes and a high number of self-collisions (Figure 8(f)).

All these benchmarks are represented in terms of the position of triangle vertices at discrete simulation time steps. Our algorithm performs continuous collision detection between these simulation time steps, and computes the first-time-of-contact.

4.3 Performance

Figure 9 highlights the performance of our algorithm on different benchmarks. These results show that collision-streams works well on different GPU architectures. Even though each GPU has a different architecture and cache hierarchy, we see direct relationship between the number of streaming processors or cores with the running time. The relative performance of a benchmark appears to be proportional to the number of streaming units on different GPUs. This indicates that our approach can exploit the large-scale parallel capabilities of current GPUs. NVIDIA GeForce GTX 480 architecture and drivers provides the flexibility to the developers in terms of choosing between larger shared memory or large L1 cache size. In our system, we chose to use a larger L1 cache size(48KB) and a smaller shared memory (16KB). As a result, we observed $1.97 - 2.62$X speedups on NVIDIA GeForce GTX 480 as comparison to NVIDIA GeForce GTX 285 for our benchmarks (Figure 12).

Figure 10 shows the running time ratios of the kernels for the Funnel benchmark on a NVIDIA GeForce GTX 480. The dynamic updating of BVTT front takes about 59.12% of the total running time. The culling and processing of feature pairs takes approximately 31.56% of the total running time. Other tasks, including updating bounding volumes, refitting BVH, exact testing of orphans and feature pairs, etc., take about 9.32% of the total running time.

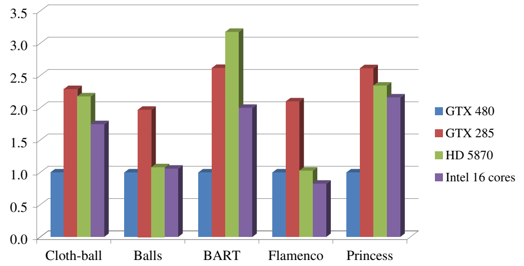

Figure 11: Performance comparison between different processors on several benchmarks: *The figure illustrates the performance of our algorithm on different GPUs and also compare the performance with a 16-core PC [Tang et al. 2010b].*

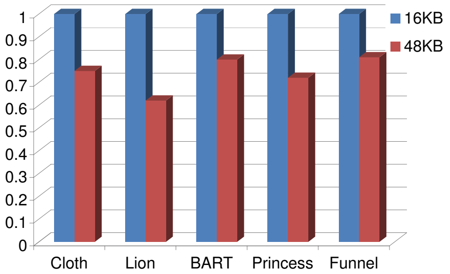

Figure 12: Performance data with smaller/larger L1 cache: *This figure shows the ratios between running times with smaller (16KB)/larger(48KB) L1 Cache for several benchmarks. For the Cloth benchmark, the running time is reduced by* 25% *as large L1 Cache is used. For the Lion benchmark, this number is* 37%.

5 Comparison and Analysis

In this section, we compare our algorithms, collision streams, with prior GPU-based algorithms and highlight some of the benefits.

Comparison: Most of the earlier methods for GPU-based collision checking performed the computations at image-space resolution [Heidelberger et al. 2003; Knott and Pai 2003; Govindaraju et al. 2003; Greß et al. 2006]. In contrast, our approach provides object-space accuracy and is quite faster as compared to these methods. Many recent methods use hybrid combinations of CPUs and GPUs to perform these queries at object-space precision [Zhang and Kim 2007; Kim et al. 2009; Pabst et al. 2010]. As opposed to these methods, our algorithm is a purely GPU-based approach and there is no communication bottleneck in terms of repeated data transfer between CPU and GPUs.In [Pabst et al. 2010], GPU-based spatial hashing is used for broad-phase culling. In general, spatial hashing maps well to the parallelization capabilities of GPUs, but it is important to choose the appropriate grid size to get optimal performance. Moreover, our update cost based on BVTT front is relatively low. Recently, Lauterbach et al. [2010] describe a fast GPU-based scheme to update and traverse the hierarchies and to perform CCD and distance queries. Their approach is based on explicit balancing of work units coupled with very lightweight synchronization between cores, and maps well to GPU architectures. In contrast, our framework based on streams is more flexible and makes it easy to incorporate with culling methods based on Orphan sets and Representative Triangles. Moreover, collision-streams has a smaller memory overhead and can handle larger models.

As comparison to multi-core CPU-based algorithms [Tang et al. 2010b] (Figure 11), our collision-streaming algorithm is mainly designed for high-throughput GPU architectures. As shown in Figure 11, our GPU algorithm is faster on almost all the benchmarks. For the Princess benchmark, a speedup of 2.16X is achieved. Compare to recent GPU-based algorithm [Lauterbach et al. 2010] which uses OBB trees for CCD computation, for the Cloth-ball benchmark, a speedup of 2.04X is achieved. Compare to recent CPU/GPU hybrid algorithm [Pabst et al. 2010] (using a single GPU), for the Funnel benchmark, a speedup of 1.34X is observed.

Analysis: The collision-stream algorithm maps well to the current GPUs and we have evaluated its performance on three different GPUs with different architectures and number of streaming processors. Furthermore, it is relatively simple to combine different culling methods and optimizations into the streaming framework. This makes it possible to develop a more flexible GPU-based framework for collision and proximity queries. Other benefits of our approach include:

- By mapping the geometric and acceleration data structures and functional modules to streaming data and computation kernels, respectively, the CCD algorithm between deformable objects can exploit the parallelism on current GPUs.
- We use front tracking to perform fine-grained task decomposition, parallelize the computation by generating a large number of sub-tasks and they are performed in parallel by distributing them among stream processors.
- The recent GPUs offer two level cache hierarchy and ability to vary the relative size of shared memory and L1 cache. Our stream registration algorithm can exploit this feature to obtain higher performance. Overall, our approach offers more flexibility in terms of mapping to current and future GPU architectures. By setting the preference of larger L1 cache size, considerable speedups achieved on Fermi cards. For the Cloth benchmark, the running time is reduced by 25% as large L1 Cache is used. For the Lion benchmark, we observe 37% improvement in performance. Figure 12 shows the ratios between running times with smaller (16KB)/larger(48KB) L1 Cache for several benchmarks.
- With deferred front tracking, we are able to perform CCD on complex scenes on GPUs with limited global memory size, makes it possible to use such algorithms on complex scenes.

5.1 Limitations

Our approach has some limitations. First, even with deferred front tracking, storing the BVTT front on a GPU can require high amount of memory space. As a result, our approach is only practical for models when the BVTT front can fit into the GPU memory. Secondly, while our stream registration algorithm can support variable-length data structures, it may frequently access global memory and this can slow down the computation. Thirdly, for small models, other GPU-based algorithms that only use shared memory to collect tasks could be faster than collision-streams.

6 Conclusion and Future Work

We present a streaming CCD algorithm for deformable objects that performs hierarchy update, traversal and intersection computations on a GPU. Our approach is flexible as it abstracts the data and acceleration structures in terms of appropriate streams and can incorporate different culling schemes by formulating appropriate kernels. We present a novel parallel stream registration algorithm and use it to efficiently support variable-length data structures on the GPU. Moreover, our approach is flexible and maps well to current GPUs in terms of memory hierarchy and varying size of shared memory. In practice, our algorithm can improve the performance of CCD algorithms on current GPU architectures. We observe speedups over CPU-based multi-core algorithm and prior GPU-based algorithms.

There are many avenues for future work. We believe that we can further improve the performance of our algorithm by improved mapping to GPU architectures. Our variable length data structure can be used for other GPU-based algorithms. Finally, we will like to extend the approach to perform other proximity queries, including distance and separation queries.

Acknowledgements: We would like to thank Simon Pabst for providing the funnel benchmark, Weiwei Xia and John Owens for useful discussions, and the anonymous reviewers for their thoughtful and constructive comments. The cloth models were provided by Rasmus Tamstorf at Disney Animation. This research is supported in part by National Basic Research Program of China (2011CB302205), MOE-Intel Fund(MOE-INTEL-09-05), ARO Contract W911NF-04-1-0088, NSF awards 0636208, 0917040 and 0904990, DARPA/RDECOM Contract WR91CRB-08-C-0137, and Intel. Tang is supported in part by Natural Science Foundation of China(60803054), Natural Science Foundation of Zhejiang, China (Y1100069), Important Science and Technology Specific Projects of Zhejiang, China (2008C01059-4), and AMD China.

References

BILLETER, M., OLSSON, O., AND ASSARSSON, U. 2009. Efficient stream compaction on wide simd many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, 159–166.

CURTIS, S., TAMSTORF, R., AND MANOCHA, D. 2008. Fast collision detection for deformable models using representative-triangles. In *SI3D '08: Proceedings of the 2008 Symposium on Interactive 3D graphics and games*, 61–69.

GOTTSCHALK, S., LIN, M. C., AND MANOCHA, D. 1996. Obbtree: a hierarchical structure for rapid interference detection. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY,USA, 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. In *Proc. of ACM SIGGRAPH/EG Workshop on Graphics Hardware*, 25–32.

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

GRAND, S. 2007. Broad-phase collision detection with CUDA. In *GPU Gems 3*, 697–721.

GRESS, A., GUTHE, M., AND KLEIN, R. 2006. GPU-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum 25*, 3 (Sept.), 497–506.

HARRIS, M., SENGUPTA, S., AND OWENS, J. 2007. Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*, 851–876.

HEIDELBERGER, B., TESCHNER, M., AND GROSS, M. 2003. Real-time volumetric intersections of deforming objects. In *Proceedings of Vision Modeling Visualization (VMV'03)*, 461–468.

HORN, D. 2005. Stream reduction operations for GPGPU applications. In *GPU Gems 2*, 573–589.

HUTTER, M., AND FUHRMANN, A. 2007. Optimized continuous collision detection for deformable triangle meshes. In *Proc. WSCG '07*, 25–32.

KIM, D., HEO, J.-P., HUH, J., KIM, J., AND YOON, S.-E. 2009. HPCCD: Hybrid parallel continuous collision detection using cpus and gpus. *Computer Graphics Forum (Pacific Graphics) 28*, 7, 1791–1800.

KLOSOWSKI, J., HELD, M., MITCHELL, J., SOWIZRAL, H., AND ZIKAN, K. 1998. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics 4*, 1, 21–37.

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

LARSSON, T., AND AKENINE-MÖLLER, T. 2006. A dynamic bounding volume hierarchy for generalized collision detection. *Computers and Graphics 30*, 3, 451–460.

LAUTERBACH, C., YOON, S., TUFT, D., AND MANOCHA, D. 2006. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. *IEEE Symposium on Interactive Ray Tracing*, 39–46.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH Construction on GPUs. *Proc. of Eurographics (Computer Graphics Forum) 28*, 2, 375–384.

LAUTERBACH, C., MO, Q., AND MANOCHA, D. 2010. gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. *Proc. of Eurographics (Computer Graphics Forum) 29*, 2, 419–428.

MERRILL, D., AND GRIMSHAW, A. 2009. Parallel scan for stream architectures. Tech. rep., CS2009-14, Department of Computer Science, University of Virginia.

MORVAN, T., REIMERS, M., AND SAMSET, E. 2008. High performance GPU-based proximity queries using distance fields. *Computer Graphics Forum 27*, 8, 2040–2052.

NICKOLLS, J., AND DALLY, W. J. 2010. The GPU computing era. *IEEE Micro 30* (Mar./Apr.), 56–69.

OTADUY, M., CHASSOT, O., STEINEMANN, D., AND GROSS, M. 2007. Balanced hierarchies for collision detection between fracturing objects. In *IEEE Virtual Reality*, 83–90.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRGER, J., LEFOHN, A., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1, 80–113.

PABST, S., KOCH, A., AND STRASSER, W. 2010. Fast and scalable CPU/GPU collision detection for rigid and deformable surfaces. *Computer Graphics Forum 29*, 5, 1605–1612.

PANTALEONI, J., AND LUEBKE, D. 2010. HLBVH: Hierarchical lbvh construction for real-time ray tracing. In *High Prerformance Graphics 2010*, 87–95.

SENGUPTA, S., HARRIS, M., GARLAND, M., AND OWENS, J. D. 2011. Efficient parallel scan algorithms for many-core gpus. In *Scientific Computing with Multicore and Accelerators*, J. Dongarra, D. A. Bader, and J. Kurzak, Eds., Chapman & Hall/CRC Computational Science. Taylor & Francis, Jan., ch. 19.

SUD, A., OTADUY, M., AND MANOCHA, D. 2004. FDiFi: Fast 3D distance field computation using graphics hardware. *Proc. of Eurographics (Computer Graphics Forum) 23*, 3, 557–566.

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.

SUPERCOMPUTING BLOG, 2009. Performance of atomics. `http://supercomputingblog.com/cuda/cuda-tutorial-5-performance-of-atomics/`.

TANG, M., CURTIS, S., YOON, S.-E., AND MANOCHA, D. 2009. ICCD: Interactive continuous collision detection between deformable models using connectivity-based culling. *IEEE Transactions on Visualization and Computer Graphics 15*, 4, 544–557.

TANG, M., MANOCHA, D., AND TONG, R. 2010. Fast continuous collision detection using deforming non-penetration filters. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, 7–13.

TANG, M., MANOCHA, D., AND TONG, R. 2010. MCCD: Multi-core collision detection between deformable models using front-based decomposition. *Graphical Models 72*, 2, 7–23.

VAN DEN BERGEN, G. 1997. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools 2*, 4, 1–14.

ZACHMANN, G., AND WELLER, R. 2006. Kinetic bounding volume hierarchies for deformable objects. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, ACM, New York, NY, USA, 189–196.

ZHANG, X., AND KIM, Y. J. 2007. Interactive collision detection for deformable models using streaming aabbs. *IEEE Transactions on Visualization and Computer Graphics 13*, 318–329.

ZHOU, K., GONG, M., HUANG, X., AND GUO, B. 2010. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics 16*, 6.