GPU accelerated Convex Hull Computation

Min Tang[a], Jie-yi Zhao[a], Ruo-feng Tong[a], Dinesh Manocha[b]

[a]*Zhejiang University, China*
[b]*University of North Carolina at Chapel Hill, USA*
http://gamma.cs.unc.edu/ghull/

Abstract

We present a hybrid algorithm to compute the convex hull of points in three or higher dimensional spaces. Our formulation uses a GPU-based interior point filter to cull away many of the points that do not lie on the boundary. The convex hull of remaining points is computed on a CPU. The GPU-based filter proceeds in an incremental manner and computes a pseudo-hull that is contained inside the convex hull of the original points. The pseudo-hull computation involves only localized operations and maps well to GPU architectures. Furthermore, the underlying approach extends to high dimensional point sets and deforming points. In practice, our culling filter can reduce the number of candidate points by two orders of magnitude. We have implemented the hybrid algorithm on commodity GPUs, and evaluated its performance on several large point sets. In practice, the GPU-based filtering algorithm can cull up to 85M interior points per second on an NVIDIA GeForce GTX 580 and the hybrid algorithm improves the overall performance of convex hull computation by $10-27$ times (for static point sets) and $22-46$ times (for deforming point sets).

1. Introduction

The problem of computing the convex hull of a set of points is fundamental in computational geometry, computer graphics and shape modeling. Given a set of points in d-dimensional space, the convex hull is the minimal convex set that contains all the points. Convex hull computation is frequently used for collision detection, interference computation, shape analysis, pattern recognition, statistics, GIS, etc.

The problem of computing the convex hull of points is well studied in geometric computing. Many optimal theoretical algorithms have been proposed for low- and high-dimensional point sets. There are also many known practical methods and robust software implementations addressing this problem [1, 2, 3].

While many theoretical algorithms have been proposed for parallel convex hull computation, there is relatively little work on fast, practical algorithms that can exploit parallel cores on the GPUs. In this paper, our goal is to design practical convex hull algorithms that can exploit GPU parallelism, as most prior practical and robust algorithms [1, 4, 5, 6] to compute convex hulls are relatively hard to parallelize.

Main Results: We present a hybrid GPU-CPU based convex hull algorithm. Given a set of points, we use a novel GPU-based filter that can cull away most of the interior points that do not lie on the boundary of the convex hull. The convex hull of the remaining points is computed using well-known CPU-based algorithms such as QuickHull [5, 3]. The GPU-based filter computes a pseudo-hull in an incremental manner which involves only localized operations and maps well to GPU architectures. Our approach ensures that the pseudo-hull lies inside the convex hull of the original points and serves as a conservative bounding shape to cull away interior points. We exploit spatial and temporal coherence between successive time steps to incrementally compute the pseudo-hull for deforming point sets and thereby reduce the runtime computation. The overall interior point filter, which exploits the parallel cores of GPUs, can cull up to 85M points on an NVIDIA GeForce GTX 580, and can reduce the number of points by almost two orders of magnitude.

Furthermore, the algorithm scales almost linearly with the number of cores. The overall approach is quite simple and the GPU-based filter can be used as a preprocess with any other convex hull algorithm or software package.

The hybrid algorithm makes no assumption about the order of input points or the underlying deformation, and easily extends to higher-dimensional point sets. We have tested its performance on 3D and 4D large point sets corresponding to static and deforming point sets on

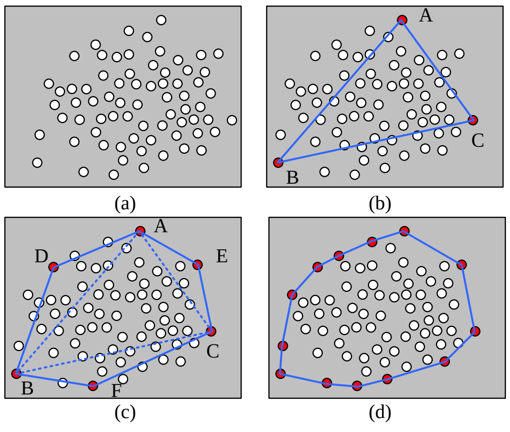

Figure 1: **2D convex hull construction:** For a 2D input point set (a), an initial simplex is constructed (b). The simplex is expanded in an iterative manner by finding the furthest point in the exterior set of each edge and replacing the original edge by two edges that are obtained based on the furthest point: e.g., AB is replaced with AD and DB (c). The algorithm terminates when there are no more exterior points (d).

different GPUs (e.g. NVIDIA GeForce GTX 285, GTX 480 and GTX 580). Our preliminary results indicate that the hybrid algorithm can speed up the computation by 27X (static point sets) and 46X (deforming point sets) as compared to an optimized CPU-based algorithm.

Organization: The rest of the paper is organized as follows: Section 2 gives a brief survey of prior work on convex hull computation. We present our CPU-GPU based hybrid algorithm and GPU-based interior point filter in Section 3. We present the implementation details and highlight the performance in Section 4. We compare our approach with prior algorithms and point out its limitations in Section 5.

2. Related Work

The construction of convex hulls essentially consists of two basic problems: locating extreme points and determining the connectivity between these points [7, 1]. Chan [6] gives a theoretically optimal output sensitive serial algorithm, though no good implementation is known.

Given n as the number of the point set and h as the number of extreme points in a point set, Amato et al. [4, 8] proposed randomized and deterministic parallel methods for constructing a convex hull in parallel and proved that the convex hull in d-dimensional space can be constructed in $O(nlogn + n^{d/2})$ time. Gupta and Sen [9] designed a parallel algorithm for 3D convex hull computation for the CRCW PRAM (Concurrent Read and Concurrent Write Parallel Random Access Machine). Its complexity is $O(\log\log n * \log h)$. Dehne et al. [10] also proposed a randomized parallel 3D convex hull algorithm for multicomputers. While these algorithms have good theoretical performance, no practical implementations are known.

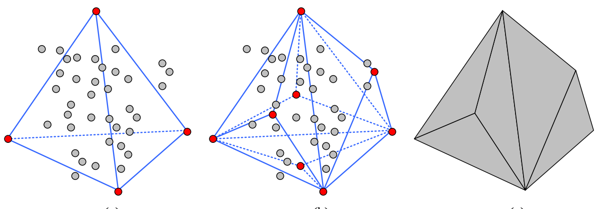

Figure 2: **Construction of pseudo-hull for culling points:** For a 3D input point set, an initial simplex is constructed (a). After a single step of expansion (similar to the 2D algorithm in Figure 1), the resulting polyhedron (i.e. the pseudo-hull) may no longer be convex (c), but can be used to cull interior points (b).

Some of the practical algorithms for convex hull algorithms are based on incremental techniques [11, 5]. One of the most widely used algorithms and software packages is based on QuickHull [5, 3], which computes the convex hull by adding one point at a time, and performs non-local operations to update the boundary. For $3D$ and higher-dimensional point sets, these non-local updates are hard to parallelize, although the 2D version of QuickHull can be parallelized on current GPUs [12]. Many techniques have been proposed in the literature to design robust implementations and handle degenerate cases [13].

Srikanth and Reddy [14] used NVIDIA GPU and Cell BE hardware to accelerate the construction of 2D convex hulls. Recently, Gao et al. [15] presented a Voronoi diagram based algorithm to construct 3D convex hull and demonstrated $3-10$X speedups over QuickHull on an NVIDIA GPU. This method seems to be limited to 3D point sets.

Some other applications, such as halfspace intersection, Delaunay triangulation [16], Voronoi diagrams, and power diagrams, are closely related to convex hull computations. Many algorithms have been proposed to compute discretized Voronoi diagrams of points and higher order primitives in 2D and 3D [2, 17, 18, 19], although these methods offer no topology or connectivity guarantees.

3. Hybrid Convex Hull Computation

In this section, we present our CPU-GPU based hybrid convex hull computation algorithm for a set of points.

Convex hull of a finite point set: The convex hull of a finite point set $P \in R^d$ is the smallest convex set that

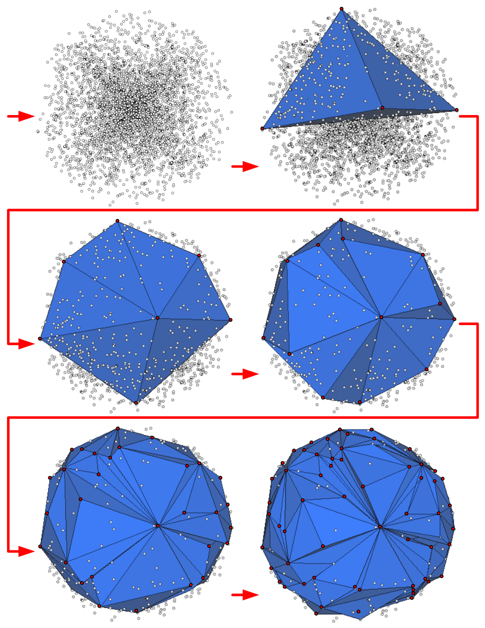

Figure 3: **Interior point culling filter:** For the input points, a tetrahedron is constructed as an initial pseudo-hull. Each facet of the pseudo-hull is updated by replacing it with three new facets. Each of these new facets is computed by connecting the edges of the old facet with the furthest point in the exterior set. All the points inside the expanding pseudo-hull are marked as interior points. After 4 iterations, most of the interior points are culled away. The pseudo-hull is incrementally updated till there are no exterior points.

contains all the points. A convex set is a set of elements such that all the points on the straight line between any two points of the set are also contained in the set. A point is *extreme* if it is a vertex of the convex hull, otherwise it is classified as an *interior point* with respect to the convex hull. During the computation of the pseudo-hull, some points that are outside the pseudo-hull are labeled as *external points*.

Our goal is to design a parallel algorithm that maps well to GPU architectures. In general, most practical algorithms for computing convex hulls perform two computations: locating the extreme points and computing the connectivity between the extreme points. In practice, it is possible to parallelize the computation of extreme points. However, most practical methods to compute the connectivity (e.g., incremental algorithms) tend to be sequential and are relatively hard to parallelize on GPU architectures.

In order to exploit the capabilities of current GPUs, we mainly use the parallel cores to cull away most of the interior points. This is based on performing fine-grained sub-tasks that can easily map to parallel cores on the GPUs. Finally, we use CPU-based algorithms (e.g. QuickHull) to construct exact convex hulls of the remaining points.

Our approach is based on some simple algorithms to compute convex hulls in 2D (Figure 1). The convex hull is constructed iteratively by expanding the boundary of a 2D polygon. The boundary of the 2D polygon is refined locally by computing the furthest point for each edge and replacing the edge by connecting the furthest point to the two vertices of the edge. The idea has been widely used for many parallel algorithms for 2D point sets [12], since the refinement of each edge is independent of the refinement of the other edges. However, the idea cannot be directly extended to $3D$ and higher-dimensional point sets. For example, as shown by Figure 2, even after a single step of expansion (i.e., the replacement of each facet with 3 facets by connecting the vertices of the facet to the furthest point in its exterior set), the resulting 3D polyhedron may no longer be convex (Figure 2(c)).

Although the expansion of a simplex cannot be directly used to generate a convex hull, it provides a simple mechanism to cull the interior points. We call the i-th expansion result starting from a d-dimensional simplex the i-th **pseudo-hull**, which is generated by repeatedly replacing each facet with three new facets by performing local operations. During the expansion of the pseudo-hull, all the points inside the pseudo-hull are classified as interior points. The pseudo-hull expansion stops when there are no more exterior points. All the interior points are culled away, and the points on the boundary of the pseudo-hull are used as an input to the exact CPU-based convex hull computation algorithm.

An example of the expansion of a single pseudo-hull is shown in Figure 3. For the input point set, an initial tetrahedron is constructed (as the 1st pseudo-hull), and expanded via each face to generate the 2nd pseudo-hull. This process terminates when there are no more points exterior to the pseudo-hull. The total number of iterations may be $O(\log n)$ (for average case) or $O(n)$ (the worst case), where n is the number of the input set.

Correctness: Based on the Grünbaum's Beneath-Beyond Theorem [20], all the points that lie inside the pseudo-hull cannot be the extreme points of the convex hull.

Benefits: The main benefit of this filter is that the expansion and refinement for each facet is performed independently and uses only local operations. It is straightforward to map the refinement of a pseudo-hull into GPU kernels that can be executed in parallel by GPU threads.

Robustness: The geometric operations executed on the GPU include computing the distance between a point and a facet (to locate the furthest point in its exterior set) and preforming beneath-beyond tests between a point and facet (to cull interior points and split the exterior sets). This boils down to evaluating Boolean predicates to perform the inside/outside tests. When the points are very close to the boundary of the pseudo-hull, the accuracy of these tests can vary due to the underlying IEEE floating-point arithmetic. In order to increase the accuracy of tests performed on the GPUs, exact arithmetic or floating filters might be employed, although the robustness of the overall algorithm depends mainly on the CPU-based convex hull algorithm and its implementation.

3.1. *Algorithm Overview*

Broadly speaking, our CPU-GPU based hybrid convex hull computation algorithm consists of two stages:

1. **GPU-based interior point culling:** Given a point set, the interior points are eliminated at each iteration by computing the pseudo-hull in an incremental manner.
2. **CPU-based exact convex hull computation:** All the points that are not culled are used as input to the exact CPU-based convex hull computation algorithm.

3.2. *Interior Point Culling Filter*

Algorithm 1 highlights the detailed stages of interior point culling filter. We first classify all the exterior points of an initial simplex (consisting of $d + 1$ vertices) into the exterior sets of all its facets. Next, each facet is expanded by connecting its d vertices with the furthest point of its exterior set. In this manner, a final pseudo-hull is iteratively computed until there is no point that is exterior to the pseudo-hull. The points lying on the boundary of the last pseudo-hull are used as an input to the exact convex hull algorithm.

In Algorithm 1, a point will only belong to the exterior set of one facet (i.e., it will not be shared by several exterior sets). At line 3-5 and 10-12, a point will be tested with all the d+1 facets, and classified as belonging to the exterior set of the facet that is closest to that point. If that point is equi-distant from multiple facets, we assign it to the first facet in that list. In this process, our algorithm performs the culling step in a distributed manner and no synchronization is needed.

3.3. *Mapping to GPU Architectures*

Algorithm 1 can be directly mapped onto GPU architectures. We abstract the facets and exterior sets as facet

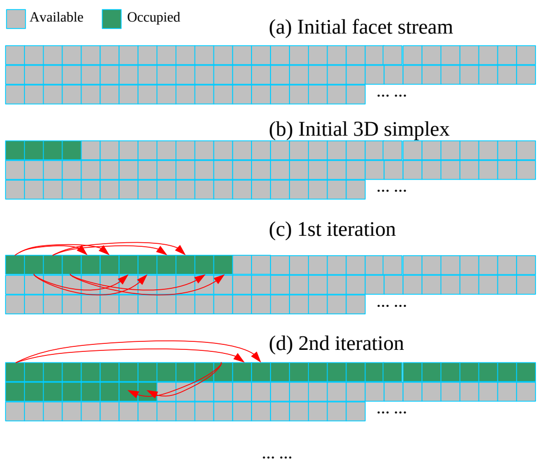

Figure 4: **Updating of facet stream:** We use a pre-allocated space in GPU memory to store the facet stream (a). For the 3D convex hull computation, an initial 3D simplex is stored (b). During each iteration, the j-th facet is replaced by three new facets, which are stored at the locations j, $K + j * 2 + 1$, and $K + j * 2 + 2$, respectively ((c)&(d)). Here K is the number of occupied cells at the last iteration.

streams and exterior set streams, respectively . Since all the facets are expanded independently, the expansion is mapped to a GPU kernel by taking the facets of the i-th pseudo-hull and all the exterior sets associated with the facets as input, and generating the $(i + 1)$-th pseudo-hull (represented by update facet streams and exterior set streams) as output. Algorithm 2 highlights the GPU-based interior point culling filter.

All input points are stored in GPU memory using stream data (i.e., a point stream.) For each iteration, all the points marked as interior points are removed from the point stream by using stream reduction with the prefix-sum operator [21]. As shown in Figure 4(a), we use a pre-allocated space in GPU memory to store the facet stream. For the 3D convex hull computation, an initial 3D simplex is stored (Figure 4(b)). For each iteration, the j-th facet is replaced by three new facets, which are saved at the locations j, $K + j * 2 + 1$, and $K + j * 2 + 2$, respectively. Here K is the number of occupied cells at the last iteration. In this way, the facet stream can be updated in parallel by multiple threads. Figure 4 (c)&(d) demonstrate the updating of the facet stream during the 1st and 2nd iterations. For some facets with empty exterior sets, no new facets will be generated, and consequently null cells are inserted into the facet stream. We use a prefix sum operator to remove these null cells at the end of each iteration.

3.4. *Extension to arbitrary dimensional cases*

In d-dimensional cases ($d > 3$), the pseudo-hulls will correspond to d-dimensional polyhedra. By testing

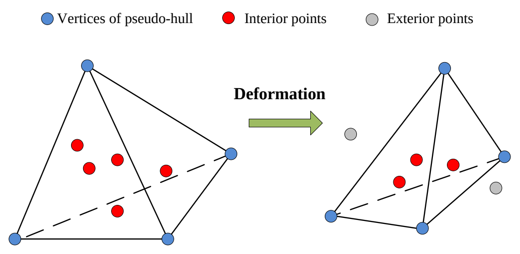

Figure 5: **The deformation of points and their corresponding simplex (part of pseudo-hull) in deforming point sets:** Some of the filtered points (the gray points) may not belong to the simplex, while others (the red points) remain inside the simplex.

the interior/exterior relationship between d-dimensional points and these polyhedra, the interior points are culled. Our algorithm has been applied to points defined in a 4-dimensional space (the 4D-Sphere benchmark in Section 4).

3.5. Deforming Point Sets

Convex hulls of deforming point sets are frequently computed in deformable simulation and data analysis, and convex hulls must be computed efficiently. For deforming point sets, we exploit the coherence between the positions of the points between two successive time steps to accelerate the construction of the convex hull by reusing the i-th pseudo-hull computed during the last time step. We do not make any assumptions on the deformation or the motion of each point.

Based on the final pseudo-hull computed during the last time step, all the points are either classified as vertices of the pseudo-hull or interior points. During the current time step, we first update the pseudo-hull with the current position of its vertices, then re-test all the marked interior points with a scanning pass whose complexity is $O(n)$, where n is the number of the input set. The scanning pass can be executed in parallel. If a point is classified as an interior point of the corresponding simplex, it can be culled. Otherwise, the point may be a boundary point or an exterior point for subsequent iterations. After this scanning pass, all the vertices of the pseudo-hull and these exterior points are used as an input for the next iteration of the GPU-based interior point culling filter.

As Figure 5 shows, all the interior points of a simplex are tested again in their new updated positions. Some of the filtered points may not lie inside the simplex and are classified as exterior points, while the others are classified as interior points.

The scanning pass maps well to GPU architectures, since all the interior tests can be performed independently. For one of the benchmarks in Section 4, the

Algorithm 1 Interior point culling filter

```
Input: Input point set
Output: Vertices of a pseudo-hull
1:  compute a simplex of $d + 1$ points
2:  for all unassigned points $p$ do
3:    for all facet $F$ do
4:      if $p$ is outside $F$ and the distance $\{p, F\}$ is minimal w.r.t $\{F\}$ then
5:        assign $p$ to $F$'s exterior set
6:      end if
7:    end for
8:  end for
9:  for all facet F with a non-empty exterior set do
10:   select the furthest point $p$ of $F$'s exterior set
11:   create $d$ new facets by connecting $p$ and the $d$ corners of $F$
12:   for all unassigned points $q$ in the exterior set of $F$ do
13:     for all new facet $F'$ do
14:       if $q$ is outside $F'$ and the distance $\{q, F'\}$ is minimal w.r.t $\{F'\}$ then
15:         assign $q$ to $F'$'s exterior set
16:       end if
17:     end for
18:   end for
19:   delete the facet F
20: end for
```

scanning pass of an 870K deforming model runs in less than 10ms, which is quite small ($< 2\%$) compared to the overall cost of convex hull construction.

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: an NVIDIA GeForce GTX 285, an NVIDIA GeForce GTX 480, and an NVIDIA GeForce GTX 580. Their specs are shown in Figure 6. We used CUDA toolkit 4.0 as the development environment. We use NVIDIA Visual Profiler to compute the kernel execution time and the data input/output time between the GPU and the host memory. The CPU version runs on a standard PC (AMD PhenomII 3.0Ghz CPU with 4GB RAM and 4 cores), though we use a single core for the CPU-based algorithm. The initial

Algorithm 2 GPU-based interior point culling filter

```
Input: Input point stream
Output: Filter point stream
 1: generate an initial facet stream $F$
 2: compute its corresponding exterior set stream $O$
 3: while any exterior set in $O$ is not empty do
 4:     for all facet $f_i$ in $F$ do
 5:         if $f_i$'s exterior set $o_i \in O$ is not empty then
 6:             replace $f_i$ with new facets
 7:             replace $o_i$ by culling interior points and split
                exterior points (if any) into new exterior sets
 8:         end if
 9:     end for
10: end while
```

GPU	GeForce GTX 285	GeForce GTX 480	GeForce GTX 580
Number of Cores	320	480	512
Memory capacity (G)	1.0	1.5	1.5
Memory clock rate (MHz)	1242	1858	2004
GPU clock rate (MHz)	648	700	772

Figure 6: **GPUs:** Three different commodity GPUs, an NVIDIA GeForce GTX 285, an NVIDIA GeForce GTX 480, and an NVIDIA GeForce GTX 580, are used to test our algorithm.

d-dimensional simplex is constructed using the Akl-Toussaint heuristic [22], i.e., the points with lowest and highest coordinates are selected to compute the initial simplex. We use the prefix-sum operator [21] to remove elements marked as deleted.

We use facet streams to represent the pseudo-hull on GPU memory, and exterior set streams to represent the exterior sets associated with the facets on the pseudo-hull. The pseudo-hull is iteratively expanded by repeatedly updating the facet streams and exterior streams locally. The expansion is terminated when the exterior set stream becomes empty.

4.2. Parallel Execution at Two Levels

The GPU's parallelism is based on blocks of threads. We divide the input point set into M subsets to generate many sub-tasks, and run the GPU-based interior culling filter on each subset in a block during each iteration of pseudo-hull computation. In our implementation, we choose M to be 64. The filter computation runs in parallel at two levels:

- **Block level:** Each block runs the GPU-based interior culling filter on a subset of input point set, filters the points, and generates the pseudo-hull for that subset.

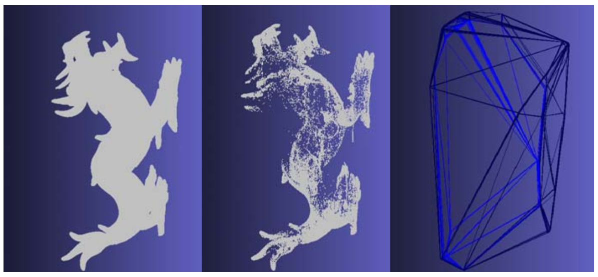

Figure 7: **Benchmark 3D-Dragon:** A dragon model with 3.6M points. The input points, filtered points, and the convex hull (with 1.8K extremal points) are displayed from left to right, respectively.

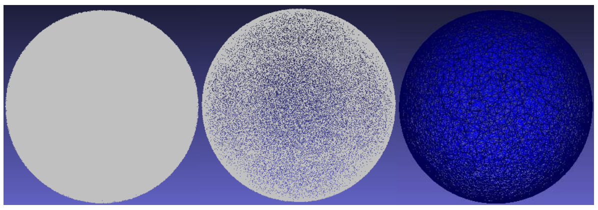

Figure 8: **Benchmark 3D-Sphere:** 8M points randomly distributed inside a sphere. The input points, filtered points, and the convex hull (with 6.8K extremal points) are displayed from left to right, respectively.

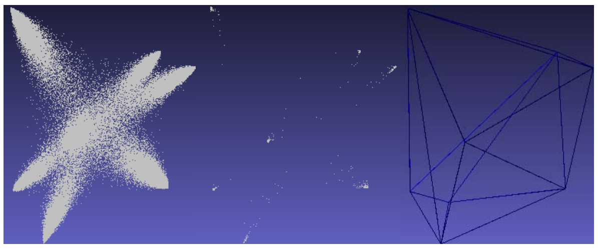

Figure 9: **Benchmark 3D-Polyhedron:** 15M points randomly distributed inside a polyhedron. The input points, filtered points, and the convex hull (with 108 extremal points) are displayed from left to right, respectively.

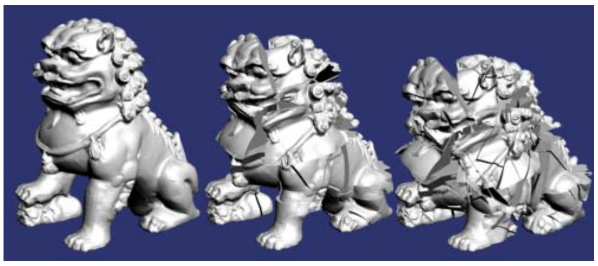

Figure 10: **Benchmark Breaking Lion:** In the benchmark, a lion model with 870K points has been crushed and is breaking into pieces. We compute the convex hull of the point set during each time step of this simulation.

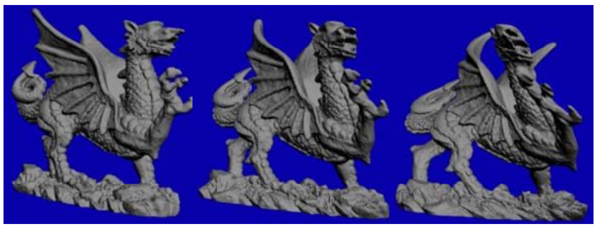

Figure 11: **Benchmark Twisting Dragon:** In the benchmark, a dragon model with 1.1M points is twisting; we compute the convex hull during each time step of this deforming model.

- **Thread level:** Because all the facets of a pseudo-hull can be processed independently, the computation and refinement of different facets are performed as different threads.

By exploiting the two-level parallelism in current GPUs, the performance of interior point culling can be improved. The filtering result may vary with different numbers of blocks, but the final output of the convex hull computation is the same. After the initialization and expansion phase, we merge the remaining points together. The remaining point set can be filtered again, or can be directly used as an input for CPU-based exact convex hull computation.

4.3. Benchmarks

In order to test the performance of our pseudo-hull filter algorithm, we used six different benchmarks, arising from different simulations that have varying characteristics.

- **3D-Dragon:** A dragon model with 3.6M points (Figure 7).
- **3D-Sphere:** 8M points randomly distributed inside a sphere (Figure 8).
- **3D-Polyhedron:** 15M points randomly distributed inside a polyhedron (Figure 9).
- **4D-Sphere:** 8M points randomly distributed inside a 4D-sphere.
- **Breaking Lion:** The 3D lion model with 870K points (Figure 10) is broken and separates into pieces. The entire simulation consists of 27 time steps and the position of the points changes during each time step.
- **Twisting Dragon:** The 3D dragon model with 1.1M points (Figure 11) is twisted, resulting in a deformation with 30 frames. The points have a new position during each frame.

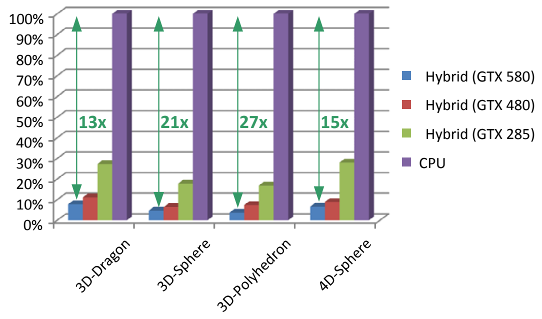

Figure 12: **Performance comparison:** Compared to the optimized CPU-based implementation, our hybrid CPU-GPU algorithm achieves 13 − 27X speedups over the CPU-based QuickHull algorithm on an NVIDIA GeForce GTX580 for all these static benchmarks.

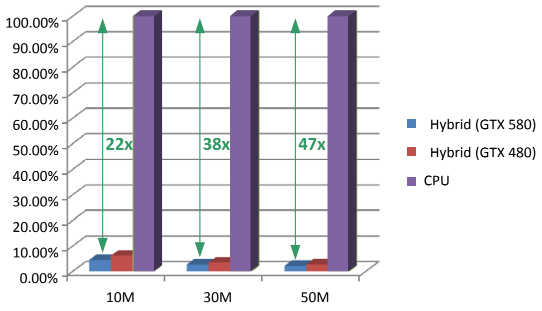

Figure 13: **Performance comparison:** By testing on 10M, 30M, and 50M points that are randomly distributed in a spherical 3D space, the hybrid algorithm demonstrates increasing speedup over the CPU-based algorithm as the size of the point set increases.

The first 3 benchmarks are used to test the performance of our filtering algorithm on static 3D point sets. The 4D-Sphere benchmark is used to test the performance on a 4-dimensional input. The Breaking Lion and Twisting Dragon benchmarks are used to test the performance on deforming point sets.

4.4. Performance

After the GPU-based interior point culling, our hybrid CPU-GPU algorithm uses QuickHull [3] to compute the exact convex hull from the filtered points on a CPU with a single thread. We compared QuickHull's performance in combination with our interior point culling filter against its performance when used in a CPU-based serial implementation without the filter.

Figure 12 highlights the performance of our hybrid algorithm on different benchmarks. These results show that our algorithm works well on three different GPUs. Compared to QuickHull [3] used alone as an optimized CPU-based implementation, the hybrid algorithm achieves 13 − 27X speedups on an NVIDIA GeForce GTX 580 for static point sets.

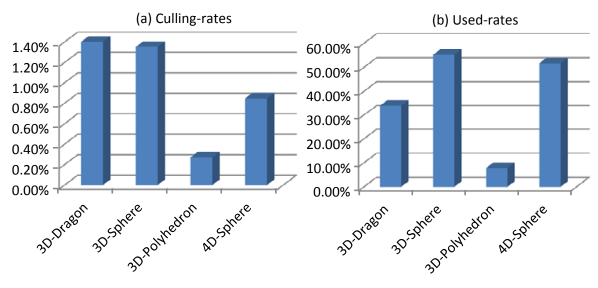

Figure 14: **Culling efficiency:** By running the pseudo-hull filter, less than 5% of the original input points are are retained, so that the size of the input to the final CPU-based computation algorithm is relatively small.

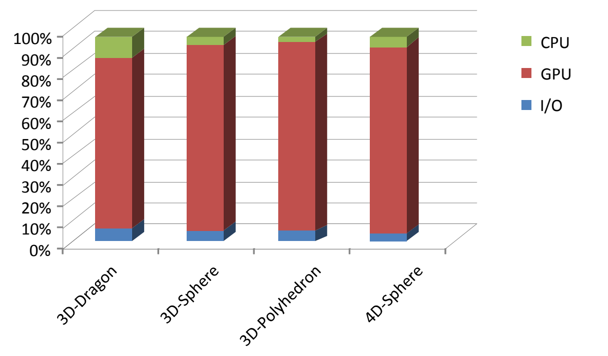

Figure 15: **Running time ratios:** the figure shows the breakdown of each phase of the hybrid algorithm on different benchmarks. Basically, this breakdown indicates that a majority of the overall time is spent in the GPU-based interior point filtering algorithm on these benchmarks. The time spent in data-transfer and CPU-based algorithm is relatively small.

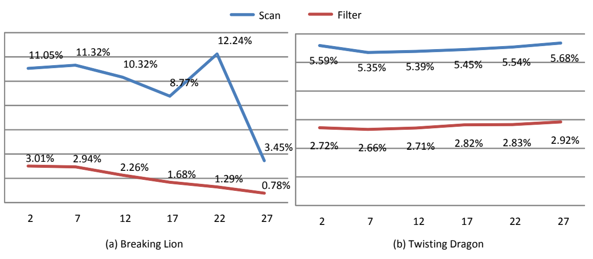

Figure 16: **Culling efficiency for deforming point sets:** By utilizing the spatial and temporal coherence between successive simulation time steps, a scan pass is performed to reuse pseudo-hulls from last simulation time step and remove interior points. Another pass is performed for interior point culling. For the Breaking Lion benchmark, only a small portion (3.45% – 12.24%) of points remain after the scanning pass. The number of points are further reduced with interior point culling, and only 0.78% – 3.01% remaining points are used for exact convex hull computation.

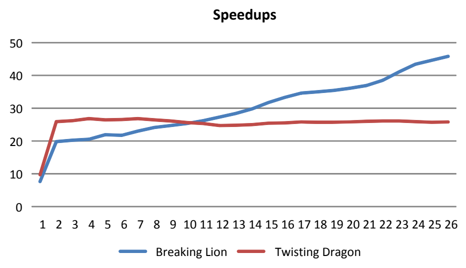

Figure 17: **Speedups for deforming point sets:** With the help of the light-weighted scan pass, the hybrid algorithm achieves up to 46X speedup for deforming point sets.

In order to highlight the scalability of the hybrid algorithm, we ran it on point sets with $10M$, $30M$, and $50M$ points that are randomly distributed within a spherical 3D space. Figure 13 shows that with the increasing number of points, computational intensity also increases; the hybrid algorithm results in increasingly larger speedups as the size of the point set increases.

Figure 14 illustrates the culling efficiency of the GPU-based interior culling filter on different benchmarks. The culling rate is computed by dividing the number of points that passed the filter by the number of input point set, while the use rate is calculated by dividing the number of extreme points by the number of points culled by the filter. As shown in Figure 14 (a), the culling rates are less than 1.4%, which demonstrates the high culling efficiency of the filter. The use rates (Figure 14 (b)) depend on the characteristics of the point sets on the input model. For example, the use rates are greater than 30% for three benchmarks. It is about 7% for benchmark 3D-Polyhedron since it has fewer extreme points.

Figure 15 highlights the ratios of each phase of the hybrid algorithm on different benchmarks. The GPU part corresponds to interior point culling, while the CPU part represents exact convex hull computation. Because of the high cull rates, the data transfer time (I/O part) represents only a small portion of the overall running time.

Figure 16 shows the culling efficiency of the scanning pass and interior point culling for a deforming point set. By utilizing the coherence between successive simulation time steps, most of the interior points can be culled for deforming point sets. For the Breaking Lion benchmark, only a small portion (3.45% – 12.24%) of points are not culled after the scanning pass. The number of points are further reduced by interior point culling, and only 0.78% – 3.01% remaining points are used for exact convex hull computation. Due to the lightweight scan pass, the hybrid algorithm achieves up to 46X speedup for deforming point sets (Figure 17).

We have tested our algorithm on a PC with 4 cores (Q6600@2.4GHz, 4G RAM) by executing the entire algorithm on these cores with OpenMP. Specifically, we run the interior point culling in parallel, and exact convex hull computation with QuickHull in serial. We observed up to 3.7X speedup on the 3D-Sphere benchmark compared to QuickHull, which runs on a single CPU core.

5. Comparison and Analysis

In this section, we compare our CPU-GPU hybrid algorithm with prior algorithms and highlight some of its benefits.

5.1. Comparison

Most of the prior CPU-based algorithms are sequential and must perform global operations, such as searching for the furthest points or updating global topology structures [5]. Although their implementations [3] can be quite efficient and robust, they are unable to exploit the computation capability of multi-core/many-core processors. Some prior GPU-based algorithms [9, 23, 14, 15] are either limited to lower dimensions (2D or 3D) or may not offer high speedups.

Recently, Tzeng and Owens [24, 25] presented a parallel convex hull algorithm based on a generalized framework for recursive divide-and-conquer on GPUs. They observe about 10x speedup comparing to a single core implementation on a CPU for $2D$ and $3D$ inputs.

Recently, Stein et al. [25] proposed a hybrid convex hull construction algorithm, CudaHull, by performing interior point culling on the GPU and concave edge swapping on the CPU. The algorithm directly computes exact convex hulls in an iterative manner and achieves about $30 - 40$X speedups as compared to a single core CPU implementation. Like CudaHull, our approach is a hybrid CPU-GPU algorithms. However, there are many differences between our approach and CudaHull. CudaHull needs to communicate between the CPU and the GPU at each iteration. As a result, our approach offers the following benefits:

- **Flexibility:** Our GPU-based interior point culling method can be combined with any exact (serial or parallel) convex hull computation algorithm.
- **Robustness:** The exact convex hull computation is performed on a CPU-based implementation, QuickHull, which is quite robust. While our GPU-based culling algorithm is susceptible to floating point errors that can arise during classification of a point with respect to the pseudo-hull, it works well on our benchmarks.
- **Generality:** Our method works for $3D$ and higher-dimensional point sets, while CudaHull only works for $3D$ point sets.
- **Efficiency:** By avoiding intensive data communication between CPU and GPU, our method can achieve higher performance.

As compared with all prior GPU-based and CPU-GPU hybrid convex algorithms, our approach has the following benefits:

- **Works well with robust CPU-based algorithms:** The GPU-based interior point culling filter can be easily integrated with other convex hull construction algorithms. Our filter can be used as the first step to reduce the number of input points for exact convex hull computation. Furthermore, it extends to higher dimensional point sets.
- **Exploits coherence for deforming points:** Our algorithm works well for deforming point sets by reusing the pseudo-hull computed during the last simulation time step.

5.2. Limitations

Our algorithm maps well to the current GPUs and we have evaluated its performance on several different GPUs with varying number of cores. Our approach has a few limitations. Our algorithm is inefficient when most of the input points are extreme points and the number of interior points is small. In this case, the extra overhead of applying our GPU-based filter can result in a slowdown as compared to a CPU based algorithm. We observed slower performance (i.e., 10% slowdown) on a 3D ball with $20M$ points on the surface. Furthermore, we assume that the entire point set fits in GPU memory. However, our approach makes no assumption about the order of interior points. As a result, it may be possible to develop an out-of-core GPU-based filter. The GPU-based filter needs to accurately evaluate the predicates to classify a given point as interior or not and is susceptible to floating point errors. In this case, we can possibly use exact arithmetic or related methods to improve the accuracy.

6. Conclusion and Future Work

We present a CPU-GPU hybrid convex hull computation algorithm that is simple to implement and can offer significant speedups. A novel GPU-based interior point culling filter is used to efficiently cull away the interior points using parallel cores. The filter has been extended to deforming point sets by using temporal and spatial coherence between successive simulation time steps. In practice, our algorithm can improve the performance of convex hull construction on commodity GPUs. We observed up to 27X and 46X speedups for static and deforming point sets, respectively.

There are many avenues for future work including development of out-of-core algorithms for handling very large data sets, and acceleration of the algorithm using the stream registration technique [26]. It may also be useful to integrate with other GPU-based algorithms that can compute the exact convex hull [15].

Acknowledgements

This research is supported in part by National Basic Research Program of China (2011CB302205), National Key Technology R&D Program of China (2012BAD35B01), NSFC (61170140), Natural Science Foundation of Zhejiang, China (Y1100069). Manocha is supported in part by ARO Contract W911NF-10-1-0506, NSF awards 0917040, 0904990, 1000579 and 1117127, and Intel. Tong is partly supported by NSFC (61170141).

References

[1] F. P. Preparata, M. I. Shamos, Computational geometry: an introduction, Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[2] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, T. Culver, Fast computation of generalized voronoi diagrams using graphics hardware, in: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, SIGGRAPH '99, 1999, pp. 277–286.

[3] qhull.org, Qhull 2010.1, http://www.qhull.org/ (2010).

[4] N. M. Amato, F. P. Preparata, An NC parallel 3D convex hull algorithm, in: Proceedings of the ninth annual symposium on Computational geometry, SCG '93, 1993, pp. 289–297.

[5] C. B. Barber, D. P. Dobkin, H. Huhdanpaa, The quickhull algorithm for convex hulls, ACM Trans. Math. Softw. 22 (1996) 469–483.

[6] T. M. Chan, Optimal output-sensitive convex hull algorithms in two and three dimensions, Discrete & Computational Geometry 16 (1996) 361–368.

[7] F. P. Preparata, S. J. Hong, Convex hulls of finite sets of points in two and three dimensions, Commun. ACM 20 (1977) 87–93.

[8] N. M. Amato, M. T. Goodrich, E. A. Ramos, Parallel algorithms for higher-dimensional convex hulls, in: Proceedings of the 34th IEEE Symposium on Foundations of Computer Science (FOCS), 1994, pp. 683–694.

[9] N. Gupta, S. Sen, Faster output-sensitive parallel algorithms for 3D convex hulls and vector maxima, Journal of Parallel and Distributed Computing 63 (4) (2003) 488 – 500.

[10] F. Dehne, X. Deng, P. Dymond, A. Fabri, A. A. Khokhar, A randomized parallel 3D convex hull algorithm for coarse grained multicomputers, in: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures, SPAA '95, 1995, pp. 27–33.

[11] K. L. Clarkson, P. W. Shor, Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental, in: Proceedings of the fourth annual symposium on Computational geometry, SCG '88, 1988, pp. 12–17.

[12] S. Srungarapu, D. P. Reddy, K. Kothapalli, P. J. Narayanan, Fast two dimensional convex hull on the GPU, in: Proceedings of AINA Workshops'11, 2011, pp. 7–12.

[13] H. Edelsbrunner, E. P. Mücke, Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms, in: Proceedings of the fourth annual symposium on Computational geometry, SCG '88, 1988, pp. 118–133.

[14] D. Srikanth, K. Kothapalli, R. Govindarajulu, P. Narayanan, Parallelizing two dimensional convex hull on NVIDIA GPU and Cell BE, in: International Conference on High Performance Computing(HiPC) 2009, Students Research Symposium, Cochin, India, 2009, pp. 1–5.

[15] M. Gao, T.-T. Cao, T.-S. Tan, Z. Huang, ghull: a three-dimensional convex hull algorithm for graphics hardware, in: Symposium on Interactive 3D Graphics and Games, I3D '11, 2011, pp. 204–204.

[16] M. Qi, T.-T. Cao, T.-S. Tan, Computing two-dimensional constrained Delaunay triangulation using graphics hardware, http://www.comp.nus.edu.sg/ tants/cdt.html (2012).

[17] I. Fischer, C. Gotsman, Fast approximation of high-order voronoi diagrams and distance transforms on the gpu, journal of graphics, GPU, and game tools 11 (4) (2006) 39–60.

[18] A. Sud, N. Govindaraju, R. Gayle, D. Manocha, Interactive 3d distance field computation using linear factorization, in: Proceedings of the 2006 symposium on Interactive 3D graphics and games, I3D '06, 2006, pp. 117–124.

[19] T.-T. Cao, K. Tang, A. Mohamed, T.-S. Tan, Parallel banding algorithm to compute exact distance transform with the gpu, in: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, I3D '10, 2010, pp. 83–90.

[20] B. Grünbaum, Measure of symmetry for convex sets, in: Proceedings of the 7th Symposium in Pure Mathematics of the American Mathematical Society, Symposium on Convexity., 1961, pp. 233–270.

[21] M. Harris, S. Sengupta, J. Owens, Parallel prefix sum (scan) with CUDA, in: H. Nguyen (Ed.), GPU Gems 3, Addison-Wesley, 2008, pp. 851–876.

[22] S. Akl, G. Toussaint, A fast convex hull algorithm, Information Processing Letters 7 (1978) 219–222.

[23] T. Jurkiewicz, P. Danilewski, Efficient quicksort and 2D convex hull for CUDA, and MPRAM as a realistic model of massively parallel computations, Max Planck Institue for Informatics.

[24] S. Tzeng, J. D. Owens, Finding convex hulls using Quickhull on the GPU, Tech. rep., University of California, Davis, `http://arxiv.org/abs/1201.2936` (2012).

[25] A. Stein, E. Geva, J. El-Sana, CudaHull: Fast parallel 3D convex hull on the GPU, accepted by Computers & Graphics (2012).

[26] M. Tang, D. Manocha, J. Lin, R. Tong, Collision-streams: Fast GPU-based collision detection for deformable models, in: I3D '11: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, 2011, pp. 63–70.