Computers & Graphics 36 (2012) 265–271

Contents lists available at SciVerse ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Applications of Geometry Processing

CudaHull: Fast parallel 3D convex hull on the GPU

Ayal Stein, Eran Geva, Jihad El-Sana*

Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

ARTICLE INFO

Article history:
Received 26 August 2011
Received in revised form
14 February 2012
Accepted 16 February 2012
Available online 17 March 2012

Keywords:
Convex hull
Parallel processing
GPU processing
CUDA programming

ABSTRACT

In this paper, we present a novel parallel algorithm for computing the convex hull of a set of points in 3D using the CUDA programming model. It is based on the QuickHull approach and starts by constructing an initial tetrahedron using four extreme points, discards the internal points, and distributes the external points to the four faces. It then proceeds iteratively. In each iteration, it refines the faces of the polyhedron, discards the internal points, and redistributes the remaining points for each face among its children faces. The refinement of a face is performed by selecting the furthest point from its associated points and generating three children triangles. In each iteration, concave edges are swapped, and concave vertices are removed to maintain convexity. The face refinement procedure is performed on the CPU, because it requires a very small fraction of the execution time (approximately 1%), and the intensive point redistribution is performed in parallel on the GPU. Our implementation outpaced the CPU-based Qhull implementation by 30 times for 10 million points and 40 times for 20 million points.

1. Introduction

For a given set of points, S, in R^d, the convex hull of S, is defined as the smallest convex set in R^d containing S, where a geometric set is convex if for every two points in the set, the line segment joining them is also in the set. The convex hull is a fundamental construction for computational geometry and has numerous applications in various fields, such as mesh generation, cluster analysis, collision detection, crystallography, metallurgy, cartography, image processing, sphere packing, and point location [1]. Other problems, such as halfspace intersection, Delaunay triangulation, and Voronoi diagrams, can be reduced to the convex hull. Fast convex-hull algorithms are useful for interactive applications, such as collision detection [2] in computer games and path planning for robotics in dynamic environments [3].

Numerous algorithms were proposed to compute the convex hull of a finite set of points with various computational complexities (see Section 2). These algorithms usually generate a non-ambiguous and efficient representation of the required convex shape. The complexity of the corresponding algorithms is usually estimated in terms of the number of input points and the number of points on the convex hull. Several algorithms have been implemented and used in practical applications. Most of them are sequential algorithms, i.e., leverage one processor of the available systems.

Over the past decade, the progress in graphics hardware development has led to a generation of powerful parallel multi-core processors: Graphics Processing Units (GPUs). These GPUs have been used to provide efficient solutions for various applications, such as particle simulation, molecular modeling, and image processing. The GPU is a massively multi-threaded architecture that includes hundreds of cores (processing elements), where each core is a multi-stage pipeline processor. Cores are grouped to generate a single instruction multi-data (SIMD) symmetric multi-processor (SM), i.e., the cores in an SM execute the same instruction on different data items. Each core has its own register set and limited local memory (usually very small), and cores within the same SM have limited shared memory.

In this paper, we present a novel parallel algorithm for computing the convex hull of a set of points in 3D using the CUDA programming model, which generates a CPU-GPU heterogeneous executable. Our algorithm adopts the well-known *Quick-Hull* approach. It starts by transferring the set of points into the video memory and uses four extreme points to generate a tetrahedron, discards the internal points, and distributes the external points to the four faces. It then iteratively refines the faces of the polyhedron, discards the internal points, and redistributes the remaining points associated with each face among its children faces. The refinement of a face is performed by selecting the furthest point from its associated points and generating three children triangles. In each iteration, concave edges are swapped, and concave vertices are removed, which leads to the generation

* Corresponding author. Tel.: +972 86477871.
E-mail addresses: ayal.stein@cs.bgu.ac.il (A. Stein), eran.geva@cs.bgu.ac.il (E. Geva), el-sana@cs.bgu.ac.il (J. El-Sana).

0097-8493/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cag.2012.02.012

of a convex polyhedron upon the completion of each iteration. Our algorithm balances the load among the cores of the available GPUs and CPUs to maximize the utilization of the hardware. Our current implementation outpaced the CPU implementation (Qhull) by 40 times for 20 million points, on average.

In the rest of this paper, we will first overview closely related work on convex-hull construction; in the following sections we will present our algorithm, implementation details, and experimental results; finally, we will draw some conclusions and suggest directions for future work.

2. Related work

Many convex-hull algorithms have been developed over the last several decades, and some have been implemented. Next, we briefly discuss closely related work on sequential, parallel, and GPU-based convex-hull construction algorithms.

Various approaches have been taken to construct the convex hull of a set of points in three dimensions (3D) sequentially. Preparata and Hong [4] developed a recursive algorithm, which is based on a divide-and-conquer scheme. Clarkson and Shor [5] introduced an incremental insertion algorithm, where the points are processed one by one with respect to the currently constructed convex hull; points within the convex hull are discarded, and external points are used to update the hull. Barber et al. [6] developed QuickHull, an efficient convex-hull-construction algorithm. It was built upon the algorithm of Clarkson and Shor [5] and starts by selecting four extreme points and iteratively adds an external point to extend the convex polyhedron until the remaining point set becomes empty. These algorithms have a time complexity of $O(n \log n)$. Randomized incremental algorithms that exhibit optimal expected performance were also proposed [7]. These process one point at a time in a random order.

Many parallel algorithms for convex-hull construction were proposed; most of them use the parallel random-access machine (PRAM) to design algorithms because of its close relationship with the sequential models [8]. Simultaneous read/write conflicts are resolved using several schemes, such as Exclusive Read Exclusive Write (EREW), Concurrent Read Exclusive Write (CREW), Exclusive Read Concurrent Write (ERCW), and Concurrent Read Concurrent Write (CRCW). Chow [9] presented a parallel convex-hull algorithm that runs at $O(\log^3 n)$ time complexity. Amato and Preparata [10] designed an $O(\log^2 n)$ time algorithm using n CREW processors. Reif and Sen [11] proposed a randomized algorithm for three-dimensional hulls in the CREW model that runs at $O(\log n)$ time using a divide-and-conquer approach on $O(n)$ processors using the CREW PRAM model. Amato et al. [12] gave a deterministic $O(\log^3 n)$ time algorithm for a convex hull in R^d using $O(n \log n + n^{\lfloor d/2 \rfloor})$ work, with high probability on the EREW PRAM. The absence of known implementation of these algorithms makes them mainly interesting from a theoretical perspective and less from a practical one.

Implementing PRAM-based algorithms on the GPU efficiently is problematic. The PRAM model is based on a shared-memory model, and the algorithms adopting this model assume that the entire convex hull is stored in the shared memory. This assumption is not valid for the current GPU and its CUDA model, because the memory model is completely different. It is difficult to synchronize the cores due to the high memory latency for random access, which makes the sharing capacity of the PRAM non-trivial for GPU implementation. In addition, the GPU performs symmetric computation, and its cores are not autonomous; thus, different-sized convex hulls cannot be processed in parallel efficiently. The GPU cores have a very small memory space and no independent cache, which leads to a high probability of cache-miss. This is also problematic because current GPUs are sensitive to cache miss due to memory latency.

Recently, Srikanth et al. [13] developed a 2D convex-hull algorithm that runs on the GPU using the CUDA programming model. The algorithm adopted the well-known QuickHull approach and provides good performance. However, it is very difficult to extend their implementation to three dimensions, which calls for a different approach. Gao et al. [14] developed a two-phase convex-hull algorithm, for three dimensions, that runs on the GPU. In the first phase, hull approximation, they utilize a digital Voronoi Diagram to approximate the convex hull of the input point set S. To this end, they adapted the Star Splaying algorithm [15] to run on the GPU and apply it to compute a convex polytope approximation. In the second phase, hull completion, they identify potential extreme points lying outside the approximation and insert them back to form the final convex hull. Then, they apply the Star Splaying algorithm on the GPU again to complete the final convex hull of S. Their algorithm works on the digital domain and manages to achieve 3–10 times the speed of the CPU-based *Qhull* algorithm.

3. Our algorithm

In this section, we present our algorithm for the parallel construction of a convex hull using the CUDA programming model, which provides a CPU-GPU heterogeneous execution environment. In CUDA terminology, the GPU is viewed as a computing device operating as a co-processor to the main CPU. The main CPU is called the *host*, and each GPU is called a *device*. A function compiled for the device is called a *kernel* and is executed on the device as many different threads. The host and device each manages its own memory; the memories of the two are often denoted as the *host memory* and *device memory*, respectively. To efficiently utilize this heterogeneous execution environment, data-parallel, computation-intensive functions should be off-loaded to the device/s.

During the initialization phase, our algorithm selects four extreme points, constructs an initial tetrahedron, removes the points that are inside the tetrahedron, and distributes the rest of the points among the four faces. The algorithm then proceeds iteratively. In each iteration, it refines the faces, removes internal points, and redistributes the remaining points in each face among its children faces. Concave edges are swapped, and concave vertices are removed to attain a convex polyhedron upon the completion of each iteration.

We refer to the convex polyhedron constructed at iteration i as the convex polyhedron, C_i, and to the points on the convex polyhedron as the *vertices*. We denote the points outside the current convex polyhedron as the *external* points and the points inside the polyhedron as the *internal* points.

The output of the algorithm is a manifold triangulated mesh, and the redistribution procedure associates every external point with a single face (Fig. 1). An external point is associated with the closest face, where the distance between a point p and a triangular face t is defined as the minimal distance between p and any point $q \in t$, i.e., $d(p,t) = \min_{q_i \in t}(\|p-q_i\|)$. For every face, t, we construct a cell, $c(t)$, which bounds the points associated with t. The cell of a face, t, is a trimmed tetrahedron (trimmed by t), whose base lies at infinity. The sides of the trimmed tetrahedron, which are denoted the *separating planes*, are defined by the three edges of t and their normals (an edge and its normal define a plane), where a normal of an edge is defined as the average of the normals of its two adjacent triangles. Fig. 2(a) depicts the structure of the cells of a triangle in two dimensions, where a cell is a trimmed triangle whose base lies at infinity. Fig. 2(b) shows the separating planes in three dimensions in

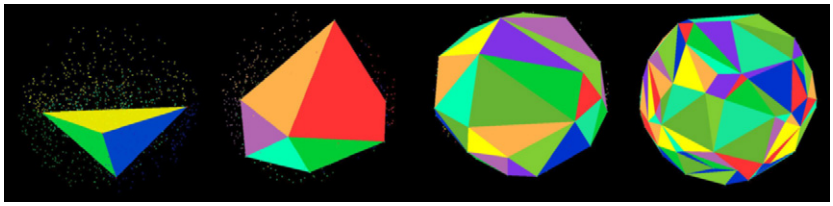

Fig. 1. Initial tetrahedron on the left, final convex hull on the right, and two snapshots during the algorithm execution.

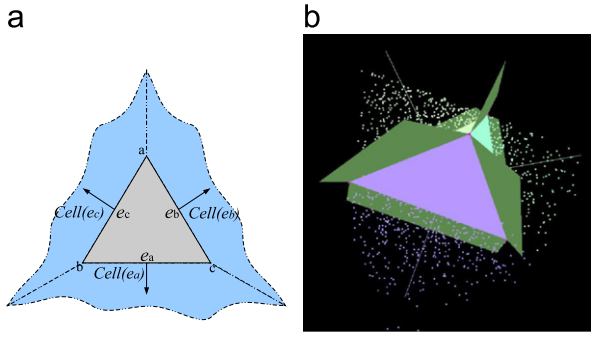

Fig. 2. (a) The cells of a triangle in two dimensions and (b) the separating planes and the triangle's cells in 3D.

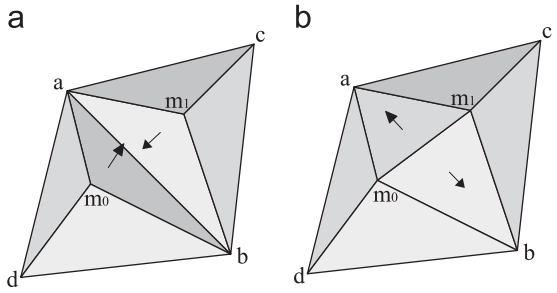

Fig. 3. (a) The refinement of two adjacent triangles $\overline{abc}$ and $\overline{abd}$; (b) the concave edge, $\overline{ab}$, is swapped by the edge $\overline{m_0m_1}$.

green. We refer to the set of points in the cell of a face t as $set(t)$. The extreme point with respect to t, denoted $extreme(t)$, is the furthest point from t in $set(t)$, i.e., $extreme(t) = \text{argmax}_{p_i \in set(t)} d(t,p_i)$.

The refinement of a triangular face t is performed by subdividing the face into three triangles, t_a, t_b, and t_c, using the extreme point of t. We refer to t as the parent triangle (face) and to the triangles t_a, t_b, and t_c as the children triangles. Fig. 3(b) shows the refinement of two adjacent triangles; the refinement of the triangle $\overline{abc}$ leads to the generation of the three triangles: $\overline{abm_1}$, $\overline{am_1c}$, and $\overline{bm_1c}$. Computing the cell of the newly generated triangles is performed by generating the separating planes for each of the newly added edges. The points in the cell of t, $set(t)$, are redistributed to the cells of its children triangles. The refinement is a local procedure, which means that an extreme point, p, with respect to a face is not necessarily an extreme point with respect to the entire point set. In such a case, p is not on the final convex hull and should be removed during the progress of the algorithm.

Unfortunately, the refinement of two adjacent faces may concave their common edge, e, which is made convex by swapping e with the edge connecting the extreme points of the two faces, as shown in Fig. 3. The swap operation manages to fix the convexity of e, but it may concave other edges, which are also fixed using swap operations. We refer to the first swap as *regular swap* and the following swaps as *complementary swaps*. A sequence of swaps may evolve around a concave vertex, whose degree is reduced after each complementary swap. When the degree of concave vertex reaches three, we remove it and its three adjacent faces and replace it with one triangle, as shown in Fig. 4. Alternatively, we could perform one additional swap and remove two congruent triangles and update the adjacency of the congruent edges (see Algorithm 1).

Swap operations are applied to the constructed convex polyhedron, and point redistribution is performed on the external points. The regular swaps are very light operations, and the complementary swap is usually performed on a small fraction of the currently constructed edges, while the point distribution is performed on the entire external point set. This observation motivated the design of our CPU-GPU algorithm, whose flow scheme is depicted in Fig. 6.

Our algorithm relies on a simple local geometric point-plane test to determine the position of a point with respect to a plane, which is used to decide whether a given point is inside or outside the constructed polyhedron and to examine the convexity of an edge or a vertex. Swap operations may lead to foldovers and self-intersections, which complicate the point-plane test and disturb their locality. For that reason, we apply only valid swaps, which are swaps that do not lead to foldovers or self-intersections. The validity of a swap is determined by the configuration of the edge and its adjacent triangles; a swap is invalid if one of the vertices of the edge is beneath the opposite swapped triangle. Fig. 5 shows an invalid swap of the edge $\overline{ab}$: the vertex a is beneath the triangle bcd.

3.1. Algorithm design

The point set is copied from the main memory to the device memory (global memory). The device selects four extreme points: p_0, p_1, p_2, and p_3. For example, p_0 and p_1 are the minimum and maximum along the x-axis, respectively, p_2 is the furthest point from the line $\overline{p_0p_1}$, and p_3 is the furthest point from the plane determined by p_0, p_1, and p_2. The four extreme points are sent to the host, which constructs the initial tetrahedron and the cells for each face. The face and separating planes, which determine the cells of each triangle, are sent to the device and stored on the small but fast cached constant memory. The device then distributes the points among these cells and determines the extreme point for each cell.

The algorithm then proceeds iteratively until the external point's set becomes empty. In each iteration, the host uses the extreme points to refine the faces of the polyhedron, swap concave edges, and remove concave vertices. The separating planes are copied to the device memory and used to redistribute the points among the cells. The device completes the iteration by computing the extreme points in each cell.

3.2. Algorithm complexity

Because the algorithm includes two modules, one runs on the host, and the other runs on the device; we analyze their

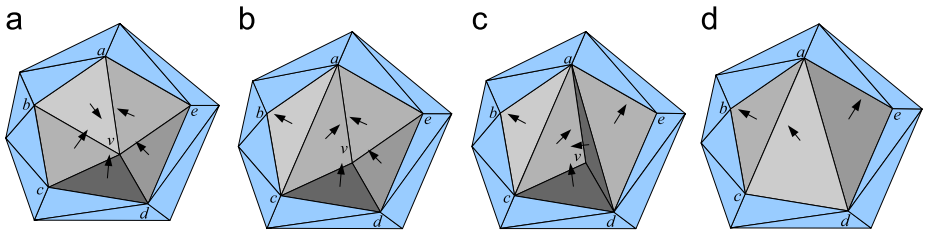

Fig. 4. A sequence of complementary swaps and the final concave vertex removal.

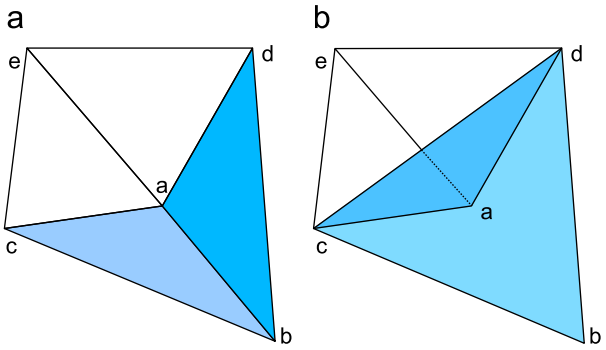

Fig. 5. Foldover: (a) before applying the swap, (b) after swapping the edge $\overline{ab}$ with the edge $\overline{cd}$.

complexities separately. Let us assume that the input includes n points and that the constructed convex hull includes k vertices.

The complexity of the module that runs on the host is tight-output sensitive. Let us assume that all the faces are refined at each iteration and that the number of removed vertices is negligible. For a tetrahedron with triangular faces $F=2V-4$, i.e., $F\approx 2V$, in each iteration, the number of faces triples, which means that the host module performs $O(\log k)$ iterations and processes $4,\ldots,k$ vertices in these iterations. That leads to $O(k)$ time complexity.

To estimate the complexity of the module that runs on the device, let us assume that the device includes p cores, i.e., it can execute p threads in parallel. Because we took the QuickHull approach, the percentage of discarded points at each iteration is similar (up to constant), which leads to $O(n\log n)$ complexity [6]. However, in our approach, the remaining points are processed by the p cores at $O((n/p)\log n)$ time complexity.

3.3. Algorithm correctness

To prove the correctness of the algorithm, it is enough to show that the constructed polyhedron at the end of each iteration is convex.

Claim 1. *The constructed polyhedron at the end of each iteration is convex.*

Proof. Without a loss of generality, we assume that no four points are coplanar. We prove the claim by contraction.

Let us assume by contradiction that the resulting polyhedron is not convex, which means that there is a concave point, p, on the surface of the polyhedron. The point p can be either on a vertex, a face, or an edge. If p is within a face, there is a concave point on an edge of that face, and if p is on a vertex, there is a concave edge adjacent to this vertex. In these two cases, there is a concave point on an edge.

If p is on an edge e, this implies that e is concave, which contradicts swapping all concave edges. The absence of four coplanar points prevents applying swaps in a circle. □

4. Implementation detail

In this section, we discuss in detail the implementation of our algorithm on the CUDA programming model. The CUDA device includes various memory types, which vary in capacity, access mode, and response time. To efficiently utilize the available computation power of the device, it is important to determine where to place the various data structures.

Algorithm 1. CudaHull Algorithm: CPU module.

```
Initialize();
while !empty(ExternalPoints) do
  for each face do
    if !empty(set(face)) then
      Refine(face, Extreme[face]);
  for each edge do
    if Concave(edge) and Valid_Swap(edge) then
      Swap(edge);
      while Complementary_Swap_Required() do
        Complementary_Swap();
        if Two_Triangles_Congruent then
          Discard_Triangles();
          Discard_Common_Vertex();
  GPU.DistributePoints();
  // These functions are implemented using CUDPP
  GPU.CompactIndexArray();
  Extreme=GPU.getExtremePoints();
```

The host places the input points on the global memory, which has a high latency compared to GPU registers. To avoid changing the order of the points, we maintain an array of indices, *index*[], which is initialized to the order of the input points, i.e., $index[i]=i$. During the execution, the internal points are moved to the tail of the array such that its head includes the vertices and the external points. In addition, we maintain an array, *associate*[], in the global memory, which stores the face associated with every point. The three arrays are maintained by the device and stored in its global memory. For each point, the device computes the distance from its associated face and stores it in the *distance*[] array, which is used to determine the extreme points in the array *extreme*[] (indexed by face *id*). The array *extreme*[] is transferred to the host to be used for refinement. In each iteration, internal points are discarded or invalidated by assigning zero to the corresponding index in the *isvalid*[] array.

The host maintains the currently constructed convex hull as a triangle mesh with adjacency to enable efficient edge swap. It avoids swapping edges that lead to foldover and swaps them later by complementary swap (in case they still need to be swapped).

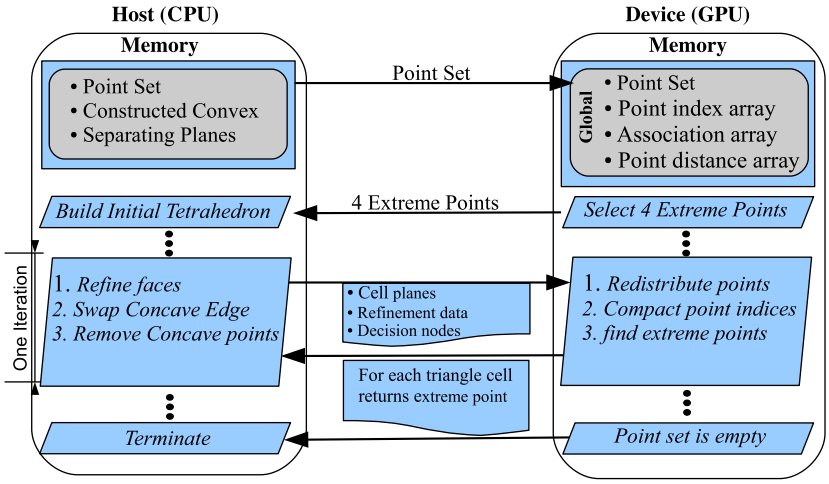

Fig. 6. The parallel construction of the convex hull and the distribution of work between the CPU and the GPU.

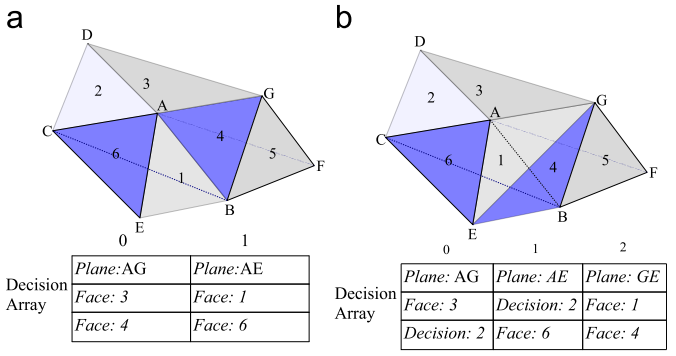

Fig. 7. The decision array: (a) after applying two disjoint swaps and (b) after adding a third overlapping swap.

From a practical standpoint, the host maintains the connectivity, while the geometry is stored in the point list. The host computes the separating planes and face planes and sends them to the device, which uses them as the boundary of the triangles' cells. Each plane is represented as four scalars (normal and offset).

During the refinement procedure, the host records refinement data for each face. These data include a flag that determines whether the face was refined or not, and for refined faces, it records the index of the three children faces. The recorded information is stored in the *refinement*[] array, which is passed to the GPU kernels to be used for point redistribution (see Algorithm 2).

Multiple instances of the redistribution *kernel* (function on the device) are executed on the GPU, each as one thread. Each thread processes one point, p_i, and assigns to *associate*[*i*] the index of the associated faces, i.e., $associate[i]=f_i$, where $p_i \in set(f_i)$. The redistribution of the points of a face to its children faces involves a comparison against two of the three separating planes within the triangle. However, the redistribution for faces adjacent to a swapped edge, *e* (regular swap and complementary), requires processing the points on the cells of the two parent faces adjacent to *e* to determine the points on its adjacent cells. To overcome this limitation, we encode the swaps in a *decision array*, which is generated by the host and used by the device. An entry in the decision array includes the index of the separating plane and defines the swap and the index of the two faces involved in the swap. In case one of the adjacent faces is involved in an additional swap, the index of that faces is replaced with a decision entry index. Fig. 7 depicts encoding three swaps in a decision array. It is important to note that the size of the decision array at each iteration is very small, e.g., in our experiments, the average size at each iteration was 88 and did exceed 200 for 10 million points, on average.

Algorithm 2. DistributePoints: GPU module.

```
parent_face=associate[index[p]];
refinement=getRefinementArray(parent_face);
decision=getDecisionArray(parent_face);
new_face=getFace(refinement, decision, index[p]);
if new_face==-1 then
  isvalid[index[p]]=0;
else
  associate[index[p]]=new_face;
distance[index[p]]=distance(index[p], associate[index[p]]);
```

CUDA distribution includes a Data Parallel Primitives library (CUDPP) that addresses various parallel procedures, such as summing, compacting, and sorting arrays. The *isvalid*[] array is used by these procedures to ignore invalid entries. To find extreme points, we sort the points first by their face *id* and then by their distances from that face. CUDPP sort uses 32 bit keys for 32 bit values, which forces us to combine the face *id* and the distance from that face in one 32 bit key, and the 32 bit value is used to store the index of the point. We use the higher $\lceil 32-\log_2(NumOfFaces)) \rceil$ bits of the key to store face *id* and quantize the distance in the remaining bits. In general, the number of faces and the distance from face exchange rules: in the early iterations, the number of faces is small and distances are large, and toward the final iterations, the number of faces is large and the distances are small. We have not experienced any limitation on the tested datasets (up to 20 million). *nVIDA* has just released a new library, *Thrust*, which supports sorting by multiple keys. This development obviates the need for the above mentioned bit layout for sorting.

5. Experimental results

We have tested our algorithm on various datasets of different sizes and produced encouraging results. Next, we present these results.

We have compared the performance of our algorithm with that of Qhull[1] using the same machine, which features an Intel *I7*

[1] http://www.qhull.org

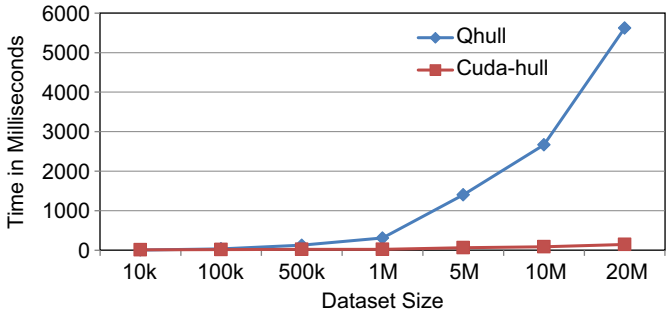

Fig. 8. The run time of our algorithm on the GPU against Qhull on the CPU using the same datasets and the same machine.

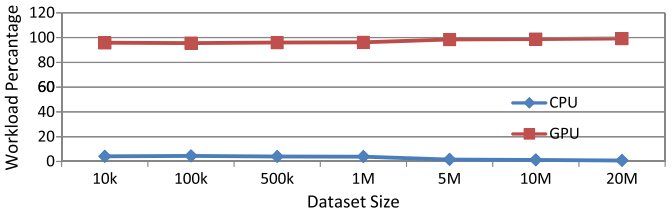

Fig. 9. The distribution of computation load between the CPU and the GPU.

Table 1
The run time (in milliseconds) performance of QHull and CudaHull.

Size	Qhull	CudaHull				Speedup
		Total	CPU	GPU	CPU (%)	
10 K	4	11	0.46	10.8	4.1	0.35
100 K	34	16	0.72	15.5	4.47	2.1
500 K	125	21	0.84	20.2	3.98	5.95
1 M	308	23	0.90	22.1	3.93	13.4
5 M	1400	65	1.05	63.9	1.62	21.5
10 M	2666	89	1.12	88.0	1.26	29.89
20 M	5622	143	1.12	142.2	0.78	39.23

processor with 12 cores and 8 GB of memory. The graphics hardware includes *nVidia GeForce GTX*580 with 1.5 GB of RAM and 512 cores.

We randomly generate point sets of different sizes that range from 10 thousands to 20 million points. For each size, we generate 10 different datasets and average their run time. Fig. 8 compares the run time of our algorithm against Qhull. As observed, our algorithm is 29 times faster for 10 M points and approximately 40 times faster for 20 M points. It also important to note that our algorithm extends its acceleration factor as the datasets grow larger.

Our heterogeneous algorithm relies on a close interaction between the CPU and GPU. We have experimented with the computation load of each processor and have found that most of the computations, approximately 99%, are performed on the massively parallel multi-threaded GPU, and that only a small fraction, approximately 1%, is performed on the CPU. Fig. 9 shows the distribution of the computation load on each of the processors for the various datasets. We use the time spent in each processor as an indicator of the processor's computation load.

The CUDA programming model provides grids and blocks to address scalability (running on different numbers of cores). Nevertheless, we have tested the performance of our algorithm on different machines (in terms of hardware) and observed similar behavior in terms of run time and computation load (all of these machines feature Intel processors and *nVidia* graphics hardware) (Table 1).

Fig. 10 shows several snapshots of the process of computing the convex hull of the Bunny model. The internal and external points are shown in red and brows, respectively. Fig. 11 and Table 2 show the final convex hull and the construction time, respectively, of several models from the Stanford 3D Scanning Repository. As observed, our algorithm handles degenerate cases, which often occur in these models, correctly.

It is also important to note that curved extreme patches, which are on the convex hull, require many triangles, i.e., when the number of points on the convex hull is large with respect to the input dataset size. In such cases, the construction time increases as a result of

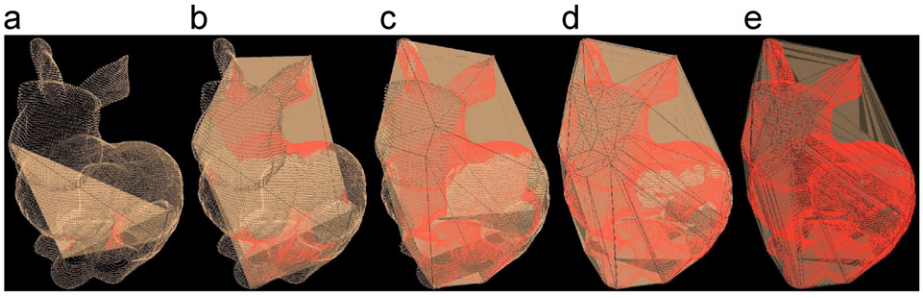

Fig. 10. The convex hull of the Bunny model: (a) initialization, (b) first iteration, (c) second iteration, (d) fourth iteration, and (e) final convex hull.

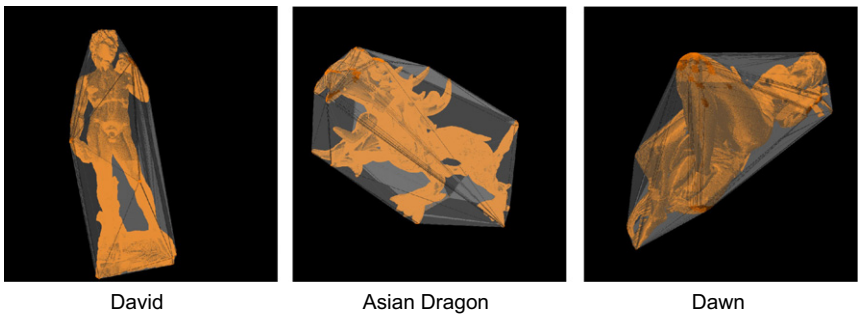

Fig. 11. The convex hull of various models from the Stanford 3D Scanning Repository.

Table 2
The run-time (in milliseconds) performance of QHull and CudaHull on real models. The size column shows the number of points in millions.

Model	Size	Qhull	CudaHull	Speedup
Buddha	2.8	642	36	18
Dragon	3.6	749	42	18
David	3.6	1073	64	17
Dawn	3.4	810	68	12

transferring large amount of data between the CPU and the GPU and the increase in the fraction of processing performed by the CPU.

6. Discussion

As we mentioned in Section 2, the proposed parallel algorithms for convex-hull construction that are based on the PRAM computation model do not seem to map well to the current GPU architecture and CUDA computation model mainly because of the fundamental difference in the memory model and the interaction between the processes running on different processors. This observation leaves the gHull [14] as the only relevant algorithm for comparison.

The run-time complexity of our algorithm, CudaHull, and that of gHall [14] are similar, because both run on $O((n/p)\log n)$ time complexity for n input point on p cores. However, the run-time complexity alone does not provide a complete picture because of the omission of constant factors, which encode the memory latency and the processing time of the various operations. This complete picture is of great importance for practical algorithms that are expected to deliver their results in a timely manner. Thus, these algorithms are tested on various datasets, and their time performance are measured (in seconds). Designing a parallel algorithm for convex-hull construction that runs on the GPU aims to reduce the actual running time of various applications that utilize convex hull to interactive time rates (30 frames/s). To demonstrate this, we provided experimental results section, which we believe provides a clear picture concerning the performance of our algorithm.

7. Conclusion

We have presented a 3D convex-hull-construction algorithm using the CUDA programming model. It is similar to QuickHull, in that it starts with an initial polyhedron constructed using four extreme points, discards inter- nal points, and redistributes the remaining points among the faces. It then proceeds iteratively. In each iteration, our algorithm reines the faces, dis- cards the internal points, and redistributes the remaining points in parallel. Our experiments on various datasets of different sizes revealed that face re- finenements and edge swaps consume a very small fraction of the execution time. Our GPU-based convex-hull algorithm performs face refinement and edge-swapping on the CPU, while the heavy-computation point redistribu-tion and extreme-points selection are performed on the parallel massively multithreaded GPU. Our implementation outpaced the CPU-based Qhull implementation by 30 times for 10 million points and 40 times for 20 million points. Our algorithm paves the way for using convex hull in various interactive applications such as computer games and dynamic path planning in robotics. The scope of future work includes integrating the fast convex hull into interactive applications.

References

[1] Aurenhammer F. Voronoi diagrams—a survey of a fundamental geometric data structure. ACM Comput Surv 1991;23:345–405.
[2] Jiménez P, Thomas F, Torras C. 3d collision detection: a survey. Comput Graph 2001;25(2):0097–8493.
[3] Choset H. Coverage for robotics—a survey of recent results. Ann Math Artif Intell 2001;31:113–26.
[4] Preparata FP, Hong SJ. Convex hulls of finite sets of points in two and three dimensions. Commun ACM 1977;20(2):87–92.
[5] Clarkson K, Shor P. Applications of random sampling in computational geometry. Discrete Comput Geom 1989;4:387–421.
[6] Barber CB, Dobkin DP, Huhdanpaa H. The quickhull algorithm for convex hulls. ACM Trans Math Softw 1996;22:469–83.
[7] Chazelle B. Randomizing an output-sensitive convex hull algorithm in three dimensions. Technical report TR-361-92. Princeton University, Princeton, NJ; 1992.
[8] Gupta N, Sen S. Faster output-sensitive parallel algorithms for 3d convex hulls and vector maxima. J Parallel Distrib Comput 2003;63:488–500.
[9] Chow A. Parallel algorithms for geometric problems. PhD thesis. University of Illinois, Urbana-Champaign; 1980.
[10] Amato NM, Preparata FP. The parallel 3d convex hull problem revisited. Int J Comput Geom Appl 1992;2:163–73.
[11] Reif J, Sen S. Optimal parallel randomized algorithms for three-dimensional convex hulls and related problems. SIAM J Comput 1992;21:466–85.
[12] Amato NM, Goodrich MT, Ramos EA. Parallel algorithms for higher-dimensional convex hulls. In: IEEE symposium on foundations of computer science; 1994. p. 683–94.
[13] Srikanth DPR, KK, Govindarajulu R, NPJ. Parallelizing two dimensional convex hull on nvidia gpu and cell be. In: IEEE international conference on high performance computing; 2009.
[14] Gao M, Cao T-T, Tan T-S, Huang Z. gHull: a three-dimensional convex hull algorithm for graphics hardware. In: Symposium on interactive 3D graphics and games, I3D '11. ACM, New York, NY, USA; 2011. p. 204.
[15] Shewchuk JR. Star splaying: an algorithm for repairing Delaunay triangulations and convex hulls. In: SCG '05: proceedings of the twenty-first annual ACM symposium on computational geometry. ACM Press; 2005. p. 237–46.