CudaPre3D: An Alternative Preprocessing Algorithm for Accelerating 3D Convex Hull Computation on the GPU

Gang MEI[1,2], Nengxiong XU[1,*]
[1]*School of Engineering and Technology, China University of Geosciences (Beijing), 100083, Beijing, China*
[2]*Institute of Earth and Environmental Science, University of Freiburg, Albertstr.23B, D-79104, Freiburg, Germany*
{gang.mei, xunengxiong}@cugb.edu.cn

***Abstract*—In the calculating of convex hulls for point sets, a preprocessing procedure that is to filter the input points by discarding non-extreme points is commonly used to improve the computational efficiency. We previously proposed a quite straightforward preprocessing approach for accelerating 2D convex hull computation on the GPU. In this paper, we extend that algorithm to being used in 3D cases. The basic ideas behind these two preprocessing algorithms are similar: first, several groups of extreme points are found according to the original set of input points and several rotated versions of the input set; then, a convex polyhedron is created using the found extreme points; and finally those interior points locating inside the formed convex polyhedron are discarded. Experimental results show that: when employing the proposed preprocessing algorithm, it achieves the speedups of about 4x on average and 5x to 6x in the best cases over the cases where the proposed approach is not used. In addition, more than 95 percent of the input points can be discarded in most experimental tests.**

***Index Terms*—computational geometry, computer aided engineering, multicore processing, parallel algorithms, parallel programming**

I. INTRODUCTION

The calculating of convex hulls for sets of points is a fundamental issue in computational geometry, computer graphics, robotics, etc, which has been well studied over four decades. The convex hull of a set of points in 3D is the smallest convex polyhedron enclosing all the input points. Numerous algorithms have been developed for constructing three-dimensional convex hull, including the gift wrapping algorithm [1], the Divide-and-Conquer algorithm [2], the randomized incremental approach [3], the optimal output-sensitive convex hull algorithm [4], and the QuickHull [5].

In recent years, several efforts have been carried out to develop practical convex hull algorithms by exploiting the power of massively parallel computing on the GPU. For example, Srikanth, et al. [6] and Srungarapu, et al. [7] parallelized the QuickHull algorithm [5] to accelerate the calculation of convex hulls for planar point sets. Also on the basis of the QuickHull algorithm, Stein, et al. [8] proposed a novel parallel algorithm for calculating the convex hull of three-dimensional points. Tang, et al. [9] developed a two-phase CPU-GPU algorithm to speed up the computing of convex hulls of points in three or higher dimensions. Similarly, Gao, et al. developed the approach gHull [10, 11]. Other efforts for dealing with the convex hull problem on GPUs include the attempts presented in [12-15].

The finding of convex hulls for large sets of points is in general quite computationally expensive. To improve the efficiency, an effective strategy, *Preprocessing*, is usually used to filter the input points by first determining and then discarding non-extreme points.

The most common preprocessing approach is to first form a convex polygon or polyhedron using several determined extreme points, and then discard those points falling into the convex polygon or polyhedron; see such applications in [8, 9, 16]. The simplest two-dimensional case is to form a convex quadrilateral using four extreme points, and then check each point to determine whether it locates inside the quadrilateral [17]. In 3D, similar preprocessing procedure can also be performed by first finding four extreme points with min or max x, y, or z coordinates to form a tetrahedron, and discarding those points locating inside the tetrahedron [5]. Another quite recent effort for efficiently discarding interior points in 2D was introduced in [18].

In our previous work [19], we proposed a straightforward preprocessing approach termed as *CudaPre* for accelerating 2D convex hull computation on the GPU. In this paper, we extend that algorithm from 2D to 3D. We term this extended algorithm as *CudaPre3D*.

There are two basic ideas behind the CudaPre3D. The first one is that: the extreme points of a convex hull of a set of points do not alter when all points in the set are rotated in the same angle(s) along the same direction(s). The second is that: those points that locate inside a convex polyhedron formed by some determined extreme points are definitely non-extreme points, and thus can be directly discarded.

Therefore, we first find several groups of extreme points by rotating the input set of points, then form a convex polyhedron / hull using the found extreme points, and finally discard those points locating inside the convex polyhedron. We use the remaining points after the discarding to calculate the expected convex hull of the input set of points.

To evaluate the performance of the CudaPre3D, We create three groups of test data, and compare the efficiency

This research was supported by the Natural Science Foundation of China (Grant No.40602037 and 40872183), China Postdoctoral Science Foundation, and the Fundamental Research Funds for the Central Universities.

Digital Object Identifier 10.4316/AECE.2015.02005

in two cases. (1) The proposed preprocessing algorithm CudaPre3D is not adopted; and the original set of input points is directly used to calculate the convex hull. (2) The algorithm CudaPre3D is first employed to filter the original set of input points; and then the remaining points are used to find the desired convex hull.

The rest of this paper is organized as follows. Section II describes the basic ideas behind the preprocessing algorithm, CudaPre3D. Section III gives some implementation details. Section IV presents three groups of experimental results. Section V discusses the CudaPre3D. And finally Section VI concludes this work.

II. The Preprocessing Algorithm CudaPre3D

There are two basic ideas behind the CudaPre3D. The first idea is that: the extreme points of a convex hull of a set of points do not alter when all the points in the set are rotated in the same angle(s) along the same direction(s). The second is that: those points that locate inside a convex polyhedron formed by some determined extreme points are definitely not the extreme points, and thus can be directly discarded. Therefore, it is reasonable to first find several groups of extreme points by rotating the input set of points, then form a convex polyhedron using the found extreme points, and finally discard those points locating inside the convex polyhedron.

The proposed preprocessing algorithm mainly consists of three steps:

Step 1: Find several groups of extreme points for the original and the rotated sets of input points;

Step 2: Create a convex polyhedron / hull of the found extreme points;

Step 3: Discard those points that locate inside the convex polyhedron / hull.

A. Step 1: Finding Extreme Points

In the calculating of convex hulls for point sets, the input points with the min or max coordinates are definitely the extreme points of the expected convex hull. The extreme points are typically found first and then to be used in the initiating / starting step of the calculating of convex hulls. For example, in QuickHull [5], four extreme points with min or max coordinates are first found to form a tetrahedron that is used as the initial convex hull.

In CudaPre3D, we first find 6 extreme points with the min or max coordinates. Then, we move the input set of points to make the center of the set of points coincide the *Origin*, and rotate all points of the moved set of input points in 30, 45, or 60 degrees along x-, y-, or z-axis. We design the following three strategies of rotating:

(1) Strategy 1:

Step (1): rotate 45 degrees along x-axis, and then rotate 45 degrees along y-axis;

Step (2): rotate 45 degrees along y-axis, and then rotate 45 degrees along z-axis;

Step (3): rotate 45 degrees along z-axis, and then rotate 45 degrees along x-axis, and finally rotate 45 degree along y-axis.

(2) Strategy 2:

Step (1): rotate 30, 45, and 60 degrees separately along x-axis;

Step (2): rotate 30, 45, and 60 degrees separately along y-axis;

Step (3): rotate 30, 45, and 60 degrees separately along z-axis; see Fig.1.

(3) Strategy 3:

This strategy is the combination of the Strategy 1 and Strategy 2.

Fig. 1 illustrates the rotating of a set of input points in 30, 45, or 60 degrees along z-axis. Other rotations such as rotating along x- or y- axis are quite similar to the rotations illustrated in Fig.1. These rotations presented in Fig.1 are in fact 2D cases. However, the rotations in 3D can be realized by performing several 2D rotations. That is, the rotation in 3D is in fact composed of several 2D rotations.

Three Variants of CudaPre3D As described above, there are three strategies of rotating. Correspondingly, we further develop three *Variants* of the CudaPre3D. More specifically, the Variant 1 of CudaPre3D is the case where the Strategy 1 of rotating is used, while the Variant 2 and Variant 3 are the cases where the Strategy 2 and Strategy 3 are adopted, respectively.

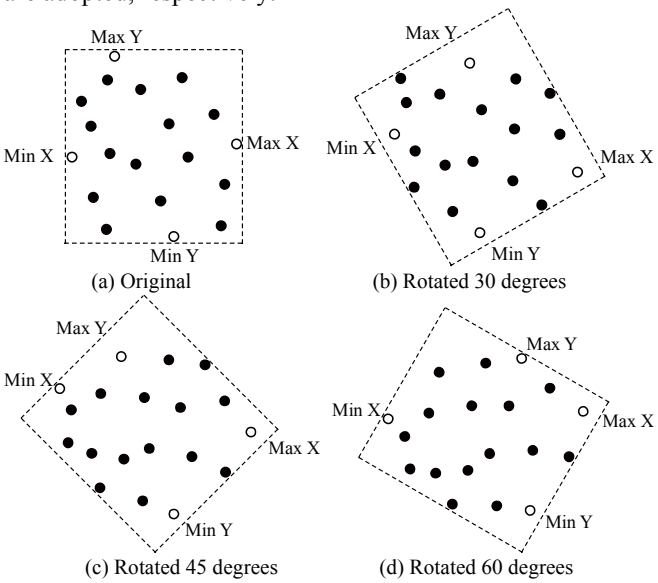


Figure 1. Original and rotated point sets and extreme points with min or max coordinates. (This figure is directly derived from the Figure 1 in [19])

B. Step 2: Creating a Convex Polyhedron / Hull

We use the incremental convex hull algorithm [20] to calculate the convex polyhedron / hull of the previously found extreme points. Note that this convex polyhedron is completely composed of triangle facets. Typically, a convex hull of a set of points in 3D is a generic convex polyhedron consisting of convex polygons. It is cannot be guaranteed that all the facets of a convex hull are triangles. Some of the facets can be generic polygons such as quadrilaterals. This is due to the fact that there are probably more than 3 points locating on the same plane. However, in CudaPre3D, we further divide such generic polygonal facets into several triangles. We perform this solution for the following two reasons:

(1) Only one type of very simple data structure, i.e., triangle, is needed to be designed;

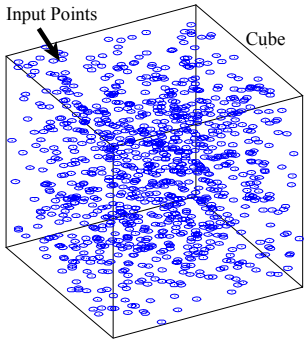
(2) It is very easy to determine the position of a point with respect to a triangle.

After calculating the convex hull of the extreme points, it is needed to check whether each points locate inside the

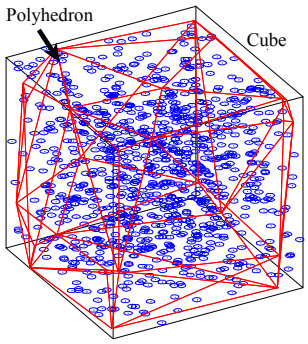
convex polyhedron / hull. Since that the determining of the position of a point with respect to a triangle is much easier than that with respect to a generic polygon. Thus, if a convex polyhedron completely consists of triangles, then it is would be very easy to check whether a point fall into this convex polyhedron. Fig.2 presents a set of input points and the convex polyhedron of the found extreme points.

C. Step 3: Discarding Interior Points

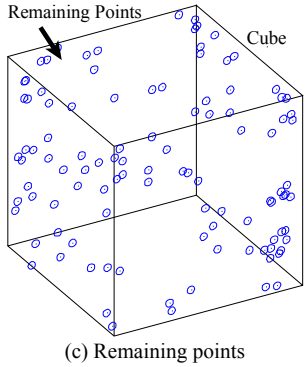
After forming the convex polyhedron / hull of the extreme points, those interior points that locate inside the convex polyhedron can be directly discarded. The key procedure in this discarding is to determine the position of each point with respect to the generated convex polyhedron. Since that the convex polyhedron is completely composed of triangles, it is only needed to determine the position of a point with respect to each triangle facet. Assuming that the normal of each facet points to the external, then if a point locate on the positive side of any facet, it is definitely non-interior points. If a point locates on the negative sides of all facets, it must be interior one, and needs to be discarded; see the remaining points of 1000 input points after discarding in Fig. 2(c).



(a) Input points



(b) Convex polyhedron



(c) Remaining points

Figure 2. The preprocessing for 1000 points randomly distributed in a cube

III. IMPLEMENTATION DETAILS

This section will presents more implementation details in the developing of the CudaPre3D. The first feature is that the implementation of CudaPre3D heavily relies on the use of the library *Thrust*. Thrust provides a series of quite useful, efficient data-parallel primitives such as parallel partition, reduction, scan, and sort. We use several of these efficient primitives for simplicity and better efficiency; see part of the implementation in Fig.3.

A. The Finding of Extreme Points

The finding of extreme points mainly includes three sub steps: (1) the moving of all points to make the center of all points coincide the origin; (2) the rotating of all points along a specific direction; (3) the finding of the points with the min or max x, y, or z coordinates.

We design a CUDA kernel, i.e., `kernelMove<<<>>>`, for moving all the points; see the code listed in line 20 in Fig.3. This moving of all points is carried out to prepare for the subsequent step of rotating all points. The coordinates of the selected local origin is the average of the min and max coordinates of all points, i.e., $P_{origin} = (P_{min} + P_{max}) / 2$. Each thread within the thread grid is responsible for calculating the new position of each point.

We also design a kernel specifically for rotating all the points in 30, 45, or 60 degrees. In each time of rotating, we only perform the rotation along a single direction such as z-axis. We realize the combined rotation by first rotating along a direction such as y-axis, and then along another direction such as z-axis. In each kernel designed for rotating, each thread takes the responsibilities to compute the rotated position of a single point.

We use the function, `thrust::minmax_element()`, to obtain the points with the min or max coordinates. Note that in this step we only need to obtain the indices rather than the coordinates of the extreme points. After finding the extreme points, we perform a checking to remove duplicate points using the function `thrust::unique()`; see line 32 in Fig.3.

B. The Creating of a Convex Polyhedron / Hull

We adopt the incremental algorithm [20] to calculate the convex polyhedron / hull of the previously found extreme points. We directly use the C++ implementation presented on page http://www.cgtools.net/treatments_convex_hull.php, and also make several minor modifications to calculate the convex polyhedron / hull. Note that the obtained convex polyhedron is completely composed of triangle facets.

C. The Discarding of Interior Points

We also design a simple CUDA kernel to discard the interior points that locating inside the convex polyhedron / hull formed in the previous step; see the lines 42 ~ 50 in Fig.3. It is also obvious that: there is no data dependency issues when checking any two points whether they fall in the convex polyhedron. A single thread can be invoked to check whether a point locates inside the convex polyhedron. After determining all the interior points, a parallel partition, i.e., `thrust::partition()` is first carried out to gather all interior points into a consecutive piece, and then a parallel copying (`thrust::copy_n`) is invoked to transfer all non-interior points from the device side to the host side for outputting; see the lines 58 ~ 66 in Fig.3.

```
int CudaPre3D(thrust::host_vector<float> &org_x, &org_y, &org_z,  // Original points
              thrust::host_vector<float> &rem_x, &rem_y, &rem_z)  // Remaining points
{
    int n = org_x.size();  // Number of input / original points
    thrust::device_vector<float> d_x = org_x;   d_y = org_y;   d_z = org_z;
    thrust::device_vector<float> d_xtmp = d_x;  d_ytmp = d_y;  d_ztmp = d_z;
    float * d_x_ptr = thrust::raw_pointer_cast(&d_x[0]);  d_y_ptr = ;  d_z_ptr = ;

    // Step 1: Rotate all points and Find extreme points
    typedef thrust::device_vector<float>::iterator floatIter;
    thrust::pair<floatIter, floatIter> extrX, extrY, extrZ
    thrust::host_vector<int> h_index;
    extrX = thrust::minmax_element(d_x.begin(), d_x.end());  // Min and Max X
    extrY = thrust::minmax_element(d_y.begin(), d_y.end());  // Min and Max Y
    extrZ = thrust::minmax_element(d_z.begin(), d_z.end());  // Min and Max Z
    h_index.push_back(extrX.first-d_x.begin()); h_index.push_back(extrX.second-d_x.begin());
    h_index.push_back(extrY.first-d_y.begin()); h_index.push_back(extrY.second-d_y.begin());
    h_index.push_back(extrZ.first-d_z.begin()); h_index.push_back(extrZ.second-d_z.begin());
    // Move to the local Origin (xc, yc, zc)
    kernelMove<<<(n + 1023) / 1024, 1024>>>(xc, yc, zc, d_x_ptr, d_y_ptr, d_z_ptr, n);

    // Rotate 45 degrees along X-axis
    kernelRotate45<<<(n + 1023) / 1024, 1024>>>(d_y_ptr, d_z_ptr, n);
    extrY = thrust::minmax_element(d_y.begin(), d_y.end());
    extrZ = thrust::minmax_element(d_z.begin(), d_z.end());
    h_index.push_back(extrY.first-d_y.begin()); h_index.push_back(extrY.second-d_y.begin());
    h_index.push_back(extrZ.first-d_z.begin()); h_index.push_back(extrZ.second-d_z.begin());
    kernelRotate45<<<(n + 1023) / 1024, 1024>>>(d_z_ptr, d_x_ptr, n);  // Along Y-axis
    kernelRotate45<<<(n + 1023) / 1024, 1024>>>(d_x_ptr, d_y_ptr, n);  // Along Z-axis
    kernelRotate30<<< >>>;  kernelRotate60<<< >>>; ... // More rotations
    thrust::sort(h_index.begin(), h_index.end());  // Sort and then Remove duplicate
    h_index.erase(thrust::unique(h_index.begin(), h_index.end()), h_index.end());

    // Step 2: Form convex polyhedron / hull
    int e = h_index.size();  float * vertices = new float[3*e];  // Coordinates
    Chull3D * poly = new Chull3D(vertices, e);  poly->compute(); // Compute convex polyhedror
    int nPoint = poly->get_n_vertices();       vertices = new float[3*nPoint];  // Points
    int nFacet = poly->get_n_faces();   int * triangles = new int[3*nFacet];    // Facets
    poly->get_convex_hull(&vertices, &nPoint, &triangles, &nFacet);  // Get points and facets
    thrust::host_vector<Point> h_point(nPoint);  thrust::host_vector<Facet> h_facet(nFacet);

    // Step 3: Discard interior points
    thrust::device_vector<Point>d_point = h_point;
    thrust::device_vector<Facet>d_facet = h_facet;
    Point * d_point_ptr = thrust::raw_pointer_cast(&d_point[0]);    Facet * d_facet_ptr = ;
    thrust::device_vector<int> d_pos(n);
    int * d_pos_ptr = thrust::raw_pointer_cast(&d_pos[0]);
    d_x = d_xtmp;  d_y = d_ytmp;  d_z = d_ztmp;
    kernelPreprocess<<<(n + 1023) / 1024, 1024>>>(d_point_ptr, nPoint, d_facet_ptr, nFacet,
                                                  d_x_ptr, d_y_ptr, d_z_ptr, d_pos_ptr, n);

    typedef thrust::device_vector<int>::iterator                    intIter;
    typedef thrust::tuple<floatIter, floatIter, floatIter, intIter>  Tuple;
    typedef thrust::zip_iterator<Tuple>                              Iter;  // zip_iterator
    Iter P_first = thrust::make_zip_iterator(make_tuple(d_x.begin(),d_y...,d_pos.begin()));
    Iter P_last  = thrust::make_zip_iterator(make_tuple(d_x.end(),d_y...,d_pos.end()));

    // Partion and then Copy
    Iter first_of_invalid = thrust::partition(P_first, P_last, is_interior_tuple());
    Tuple pos_first = P_first.get_iterator_tuple();
    n = first_of_invalid - P_first;  // Number of remaining points
    thrust::copy_n(thrust::get<0>(pos_first), n, rem_x.begin());
    thrust::copy_n(thrust::get<1>(pos_first), n, rem_y.begin());
    thrust::copy_n(thrust::get<2>(pos_first), n, rem_z.begin());
    return n;
}
```

Figure 3. The implementation of the proposed algorithm CudaPre3D

IV. RESULTS

To evaluate the performance of CudaPre3D, we compare the running time of finding three-dimensional convex hulls in two cases: (1) the preprocessing algorithm CudaPre3D is not adopted, and the original point set is directly used to calculate the convex hull; (2) the algorithm CudaPre3D is first employed to filter the original set of points, and then the remaining points are used to find the convex hull.

We have tested the proposed preprocessing algorithm by employing the famous Qhull library (www.qhull.org) on the following platform. The used machine features an Intel i7-3610QM processor (2.30GHz), 6GB of memory and a NVIDIA GeForce GTX 660M graphics card. The graphics card GTX 660M has 2GB of RAM and 384 cores. We have used CUDA toolkit version 6.0 on Window 7 Professional to evaluate all the experimental tests.

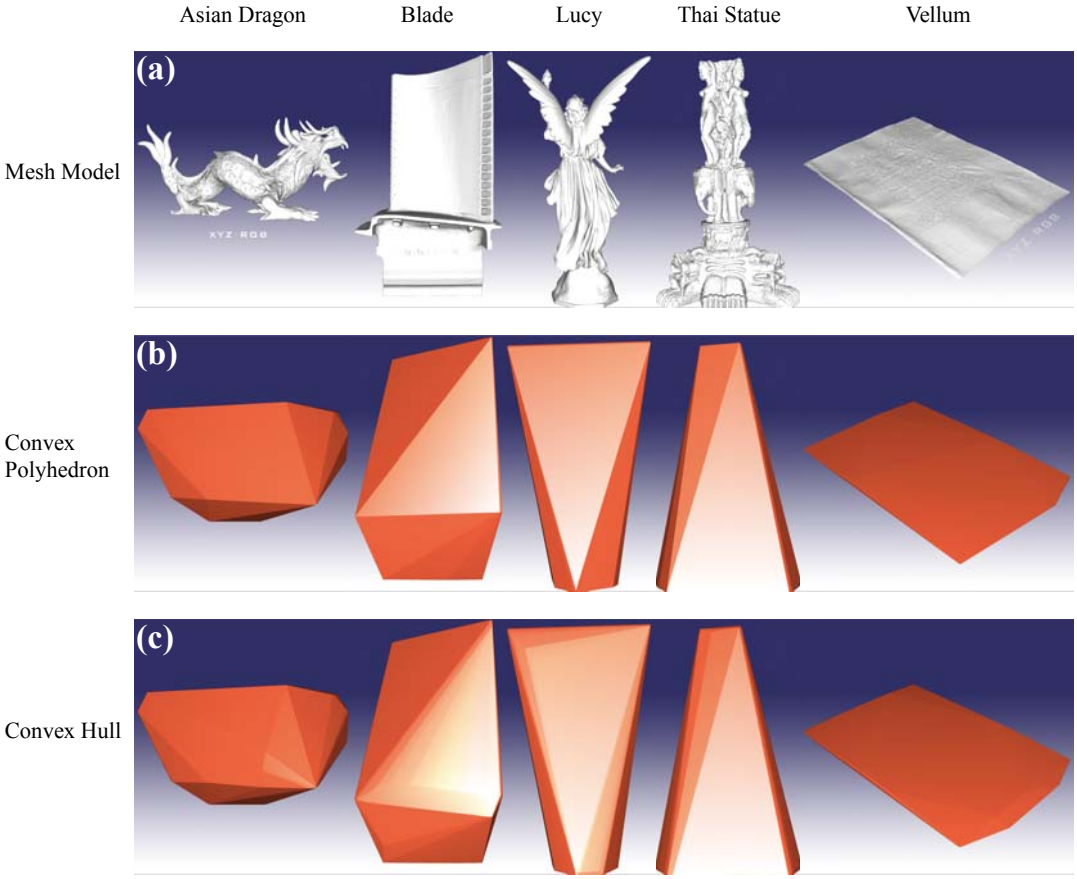


Figure 4. 3D mesh models, and the convex polyhedrons of determined extreme points and the desired convex hulls for the points derived from 3D mesh models

We have created three groups of datasets for testing. The first group includes 5 sets of randomly distributed points in a Cube that are generated using the `rbox` component in Qhull. Similarly, the second group is composed of 5 sets of randomly distributed points in a Sphere. Note that this group of test data is achieved on the basis of the first group. More specifically, we check the points in the cube of edge length 1 to determine that whether they locate inside a sphere with the radius of 0.707, and retain those points falling into the sphere as the test data. The third group consists of 5 point sets that are derived from 3D mesh models. The vertices of each mesh model are used as a set of input points. These mesh models presented in Fig. 4(a) and listed in Table III are obtained from the Stanford 3D Scanning Repository (http://www-graphics.stanford.edu/data/3Dscanrep/) and the GIT Large Geometry Models Archive (http://www.cc.gatech. edu/projects/large_models/).

Fig. 4 also provides the convex polyhedron formed by the found extreme points in the use of CudaPre3D, and the desired convex hull for each set of input points derived from those 3D mesh models.

A. Tests for Points Distributed in a Cube

We have performed the experimental tests for the points that randomly distributed in a cube of edge length 1 using three variants of CudaPre3D, i.e., the Variants 1 ~ 3; see the results listed in Tables I ~ III. Details about those variants are presented in Section II.

As introduced in Section II, there are mainly three steps in CudaPre3D: (1) the rotating of all input points and the determining of the extreme points, (2) the forming of a convex polyhedron, and (3) the discarding of interior points (i.e., those points that locate inside the previously formed convex polyhedron). Both the first and the third steps are performed on the GPU, while the second step is carried out on the CPU. To evaluate the computational workload and performance bottleneck of the CudaPre3D, we record the running time of each step in CudaPre3D. Each step of CudaPre3D has been described in more details in Section II.

The experimental results show that:

(1) Among the three variants of CudaPre3D, the first one can achieve the best performance.

For each set of points, the speedups achieved by using the Variant 1 are always greater than those by the Variant 2 and Variant 3. And similarly, when using the Variant 2, the speedup is always greater than that obtained by the Variant 3 for each set of input data. For all the three variants, the speedups are 4x on average, and 5x ~ 6x in the best cases.

(2) For each set of input points, the preprocessing procedure accomplished by performing CudaPre3D is much more computationally expensive than the calculating of the convex hull of the remaining point using Qhull.

For the relatively small size of point set, for example, the set of 1M points, the running time of Qhull is much less than that of the CudaPre3D. As the size of the input point sets increases, the running time of CudaPre3D correspondingly increases significantly, while the running time of Qhull is almost unchanged. This behavior is due to: (1) there are quite small amount of remaining points after performing CudaPre3D; and (2) the computation of the convex hull of the remaining points are quite computationally cheap.

(3) The Step 3 of CudaPre3D, i.e., the discarding of interior points, is a little more computationally expensive than the Step 1; moreover, both the Step 1 and Step 3 are far more computationally expensive than the Step 2.

For each set of the input points, the forming of a convex polyhedron for the determined extreme points always costs less than 1ms. These results indicate that the performance bottleneck of CudaPre3D does not fall into this step. In addition, when using the Variant 1, the running time of the Step 3 always much more than that of the Step 1.

Note that each step of CudaPre3D has been described in detail in Section II and Section III.

Abbreviations: **QH** means Qhull; **Spd** means Speedup.

TABLE I. COMPARISON OF RUNNING TIME (/MS) FOR THE POINT SETS DISTRIBUTED IN CUBES WHEN USING THE VARIANT **1** OF CUDAPRE3D

Size	QH	CudaPre3D + Qhull (QH)					Spd
		Total	CudaPre3D			QH	
			Step 1	Step 2	Step 3		
1M	242	65.0	22.5	0.3	27.2	15	3.72
2M	432	108.5	34.9	0.5	58.1	15	3.98
5M	1139	231.7	75.2	0.5	140.9	15	4.92
10M	2298	409.6	132.9	0.5	261.2	15	5.61
20M	4553	744.6	252.2	0.6	476.8	15	6.11

TABLE II. COMPARISON OF RUNNING TIME (/MS) FOR THE POINT SETS DISTRIBUTED IN CUBES WHEN USING THE VARIANT **2** OF CUDAPRE3D

Size	QH	CudaPre3D + Qhull (QH)					Spd
		Total	CudaPre3D			QH	
			Step 1	Step 2	Step 3		
1M	242	72.8	26.8	0.4	30.6	15	3.32
2M	432	124.3	45.3	0.5	63.5	15	3.48
5M	1139	256.1	87.5	0.6	153.0	15	4.45
10M	2298	444.5	160.3	0.6	267.6	16	5.17
20M	4553	855.1	304.9	0.6	533.6	16	5.32

TABLE III. COMPARISON OF RUNNING TIME (/MS) FOR THE POINT SETS DISTRIBUTED IN CUBES WHEN USING THE VARIANT **3** OF CUDAPRE3D

Size	QH	CudaPre3D + Qhull (QH)					Spd
		Total	CudaPre3D			QH	
			Step 1	Step 2	Step 3		
1M	242	88.8	40.3	0.5	32.9	15	2.73
2M	432	150.2	64.5	0.7	69.9	15	2.88
5M	1139	314.5	134.5	0.6	164.4	15	3.62
10M	2298	534.7	242.3	0.5	276.9	15	4.30
20M	4553	1016.4	443.6	0.6	557.2	15	4.48

B. Tests for Points Distributed in a Sphere

Tables IV ~ VI list the running time of performing Qhull with and without using three variants of the CudarPre3D for five sets of points distributed in a sphere.

Similar to the experimental results for the groups of five sets of points that locate inside a cube, (1) among the three variants, the first one can achieve the best performance; (2) CudaPre3D is much more computationally expensive than Qhull when employing the Variant 3; (3) the Step 3 of CudaPre3D (i.e., the discarding of interior points) is a little more computationally expensive than the Step 1; and both the Step 1 and Step 3 are also far more computationally expensive than the Step 2.

However, there are also several different results.

(1) The speedups

When using the Variant 2, its speedup is no longer greater but typically less than that obtained by the Variant 3. In addition, for all the three variants, the speedups are 3x ~ 4x on average, and 4x ~ 5x in the best cases.

(2) When using both the Variants 1 and Variant 2, the preprocessing procedure accomplished by employing the CudaPre3D is no longer far more computationally expensive than the calculating of the convex hull of the remaining point using Qhull.

When using the Variant 1, CudaPre3D cost only a little more running time than Qhull. In contrast, when using the Variant 2, Qhull is more computationally expensive than CudaPre3D. This result indicate that: a large number of remaining points still exist after performing the CudaPre3D. Therefore, it needs to cost much more running time to compute the expected convex hull of all remaining points using Qhull.

TABLE IV. COMPARISON OF RUNNING TIME (/MS) FOR THE POINT SETS DISTRIBUTED IN SPHERES WHEN USING THE VARIANT **1** OF CUDAPRE3D

Size	QH	CudaPre3D + Qhull (QH)					Spd
		Total	CudaPre3D			QH	
			Step 1	Step 2	Step 3		
1M	366	86.4	21.8	0.4	33.2	31	4.24
2M	673	154.8	32.8	0.5	64.5	57	4.35
5M	1701	356.1	70.7	0.5	158.9	126	4.78
10M	3799	742.5	130.2	0.6	297.7	314	5.12
20M	6651	1403.4	249.2	0.5	561.7	592	4.74

TABLE V. COMPARISON OF RUNNING TIME (/MS) FOR THE POINT SETS DISTRIBUTED IN SPHERES WHEN USING THE VARIANT **2** OF CUDAPRE3D

Size	QH	CudaPre3D + Qhull (QH)					Spd
		Total	CudaPre3D			QH	
			Step 1	Step 2	Step 3		
1M	366	140.1	47.4	0.8	29.9	62	2.61
2M	673	269.9	73.1	0.7	55.1	141	2.49
5M	1701	601.1	112.5	0.8	128.8	359	2.83
10M	3799	1216.4	191.7	0.9	242.8	781	3.12
20M	6651	2198.7	343.9	0.7	481.1	1373	3.02

TABLE VI. COMPARISON OF RUNNING TIME (/MS) FOR THE POINT SETS DISTRIBUTED IN SPHERES WHEN USING THE VARIANT **3** OF CUDAPRE3D

Size	QH	CudaPre3D + Qhull (QH)					Spd
		Total	CudaPre3D			QH	
			Step 1	Step 2	Step 3		
1M	366	123.1	40.1	0.8	52.2	30	2.97
2M	673	206.6	62.7	1.1	102.8	40	3.26
5M	1701	453.5	131.6	1.0	240.9	80	3.75
10M	3799	865.8	235.4	1.0	459.4	170	4.39
20M	6651	1638.3	426.7	1.0	850.6	360	4.06

C. Tests for Points Derived from 3D Mesh Models

This group of test data includes five sets of points that are completely derived from five 3D mesh models. The vertices of each mesh model are directly used as a set of input points, while the elements of a mesh, i.e., the triangles, are ignored. We calculate the convex hulls of these mesh models with and without using the CudaPre3D; see the experimental results listed in Tables VII ~ IX and Fig.4. Note that the performances of CudaPre3D, e.g., the speedups, are far from the expectation and satisfaction. The motivation why we still present these experimental tests in this section is to give several negative results and counter examples in the use of the CudaPre3D.

Also similar to those results for the first groups of tests (i.e., the experimental tests for the sets of points that are randomly distributed in a cube), we have observed the following behaviors:

(1) The Variant 1 of CudaPre3D can achieve the best effeciency.

For each set of input points, the speedup achieved by using the Variant 1 are always greater than those by the Variant 2 and Variant 3. In addition, when using the Variant 2 its speedup is always greater than that obtained by the Variant 3 for each set of input data. For all the three variants, the speedups are about 2x, which is far from satisfaction.

(2) For most sets of the input points, the preprocessing procedure accomplished by performing CudaPre3D is much more computationally expensive than the calculating of the convex hull of the remaining point using Qhull.

No matter which variant is selected, the preprocessing accomplished by performing CudaPre3D is much more computationally expensive than the finalizing of calculating the convex hull using Qhull for the four sets of points that are derived from the models Blade, Asian Dragon, Thai Statue, and Lucy. However, for the set of points derived from the model Vellum, the Qhull is a little computational expensive than the CudaPre3D.

There are also two differences when compared to those results of the first group of tests.

(1) The first one is that the speedups obtained by using those three variants are about 2x in most cases, while the speedups are 4x on average, and 5x ~ 6x in the best cases for the first group of tests.

(2) The Step 3 of CudaPre3D (i.e., the discarding of interior points) is no longer a little more computationally expensive than the Step 1, but much more expensive than the Step 1. In addition, both the Step 1 and Step 3 are still far more computationally expensive than the Step 2.

TABLE VII. COMPARISON OF RUNNING TIME (/MS) FOR THE POINT SETS DERIVED FROM 3D MESH MODELS WHEN USING THE VARIANT **1** OF CUDAPRE3D

Model	**QH**	**CudaPre3D + Qhull (QH)**					**Spd**
		Total	CudaPre3D			QH	
			Step 1	Step 2	Step 3		
Blade	203	90.0	20.7	0.7	30.6	38	2.26
Vellum	358	185.2	38.6	0.4	48.2	98	1.93
Asian Dragon	608	245.0	56.4	0.6	116.0	72	2.48
Thai Statue	764	262.1	74.0	0.5	152.6	35	2.91
Lucy	2184	775.2	185.5	0.6	410.1	179	2.82

TABLE VIII. COMPARISON OF RUNNING TIME (/MS) FOR THE POINT SETS DERIVED FROM 3D MESH MODELS WHEN USING THE VARIANT **2** OF CUDAPRE3D

Model	**QH**	**CudaPre3D + Qhull (QH)**					**Spd**
		Total	CudaPre3D			QH	
			Step 1	Step 2	Step 3		
Blade	203	117.7	31.2	0.9	38.5	47	1.73
Vellum	358	240.8	59.3	0.8	55.7	125	1.49
Asian Dragon	608	308.3	78.8	1.0	150.5	78	1.97
Thai Statue	764	329.4	101.2	1.0	181.3	46	2.32
Lucy	2184	1001.4	212.9	0.8	506.8	281	2.18

TABLE IX. COMPARISON OF RUNNING TIME (/MS) FOR THE POINT SETS DERIVED FROM 3D MESH MODELS WHEN USING THE VARIANT **3** OF CUDAPRE3D

Model	**QH**	**CudaPre3D + Qhull (QH)**					**Spd**
		Total	CudaPre3D			QH	
			Step 1	Step 2	Step 3		
Blade	203	116.7	37.9	1.0	47.8	30	1.74
Vellum	358	230.2	71.5	0.4	58.3	100	1.56
Asian Dragon	608	352.3	104.7	1.1	196.6	50	1.73
Thai Statue	764	404.4	132.8	1.0	240.7	30	1.89
Lucy	2184	1179.3	324.1	1.1	714.2	140	1.85

D. Effectiveness of Discarding Interior Points

The effectiveness of discarding interior points for each groups of test data are significantly varied. Fig.5 presents the percentages of remaining points after performing three variants of CudaPre3D for the groups of points distributed in a cube, distributed in a sphere, and derived from mesh models. For all the test data, the Variant 2 of CudaPre3D obtains the worst effectiveness; and both the Variant 1 and the Variant 3 are much more effective when compared to the Variant 2. In addition, for the points randomly distributed in a cube, the effectiveness of employing CudaPre3D is the most significant. That is, relatively much less remaining points exist than those for the points distributed in a sphere and the points derived from mesh models.

Fig.5 also indicates that: (1) for the points distributed in a cube or in a sphere, the percentages of remaining points for the sets with different sizes, for example, 1M and 10M of input points, do not varies dramatically; (2) for the points derived from mesh models, the percentages of remaining points for the sets of points obtained from different mesh models are model-specific. The percentages for the sets derived from different models, e.g., those points from the model Vellum and Asian Dragon, varies dramatically.

V. DISCUSSION

A. Comparison

Tang, et al. [9] developed a GPU-based interior points filtering approach. They first created a tetrahedron as the initial pseudo-hull, and then expanded the pseudo-hull by repeatedly replacing each facet with three new facets. In the expanding, those points locating inside the pseudo-hull were determined as interior points. The iterative procedure of expanding terminated when no more exterior points can be found. After discarding all the interior points, the remaining points were used to compute the convex hull on the CPU.

Compared to Tang's filtering algorithm, the most obvious advantage of our algorithm is that: it is quite easy to implement and well suited to map to the GPU architecture.

The first reason why our algorithm is much easier to implement is that: it doest not have data dependencies in the forming of a pseudo-hull (i.e., a polyhedron). The pseudo-hull in our algorithm is generated in one pass, and does not need to be iteratively expand. It is not needed to update / expand the pseudo-hull in subsequent steps.

In contrast, Tang's algorithm first creates a tetrahedron, and then expands the tetrahedron into much more complex polyhedron. It is obvious that: the forming of the $(i+1)^{th}$ pseudo-hull can only be carried out after the completion of the forming of the i^{th} pseudo-hull. Therefore, there exist data dependencies in two passes of forming. When mapping the forming of pseudo-hull to the GPU architecture, due to the data dependencies between different passes of forming, it requires several explicit locking and concurrency control techniques. Thus, it is much more difficult to implement.

The second reason is that: the data structures required in our algorithm are much simpler than those needed in Tang's algorithm. In our algorithm, we only allocate discrete arrays to store (1) the coordinates of the input points and extreme vertices and (2) the indices of points in the triangular facets of only one polyhedron. Obviously, there are no complex data structures or arrays of structures.

In contrast, due to the fact that Tang's algorithm needs to iteratively expand the pseudo-hull, it is needed to develop much more complex data structures for representing the expanding polyhedron. In other words, in our algorithm, the pseudo-hull can be considered as "*Static*". We can directly use the discrete arrays to represent the pseudo-hull. In Tang's algorithm, their pseudo-hull is "*Dynamic*" since it needs to be iteratively expanded. Thus, several complex data structures are needed in Tang's algorithm.

However, our algorithm may be not as effective as Tang's algorithm. This is because that the pseudo-hull generated in our algorithm is probably not as "ideal" as that by Tang.

The basic idea behind those two preprocessing algorithm is to discard those points locating inside a pseudo-hull. The pseudo-hull is an approximation of the exact convex hull. It is obvious that: the more is the pseudo-hull close to the exact convex hull, the more are the interior points that can be discarded (i.e., the better is the effectiveness of filtering).

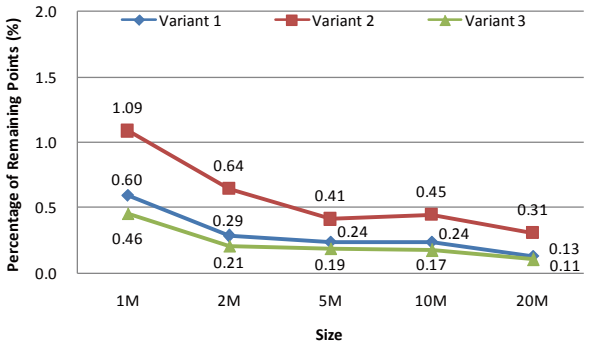
In our algorithm, we use those extreme vertices with the min or max x, y, or z coordinates to form a pseudo-hull. It is not able to guarantee that our pseudo-hull can be well close to the expected convex hull. In Tang's algorithm, they first create a tetrahedron as the initial pseudo-hull. And with the iterative expanding of the pseudo-hull, the pseudo-hull becomes more and more close to the exact convex hull. And, more and more interior points can be found and discarded.

In short, the pseudo-hull generated in Tang's algorithm is probably more close to the expected convex hull than that by our algorithm. Therefore, Tang's algorithm may achieve better effectiveness than our algorithm.

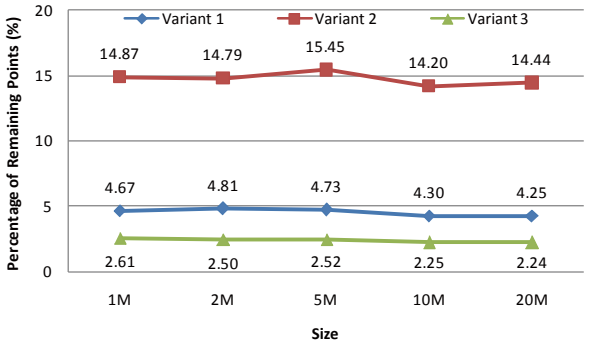
Note that both our algorithm and Tang's algorithm have the same limitation [9]: if most of the input points are extreme vertices, both of the algorithms are even slower than the CPU-based algorithms due to the time wasted on the preprocess step on the GPU.

B. *Effect of the Strategies of Rotating*

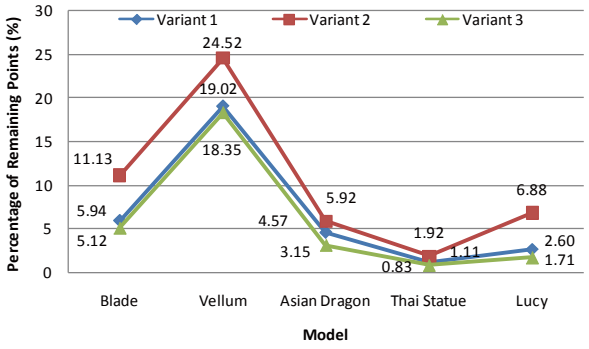
We have tested three strategies of rotating for determining extreme points and discarding interior points. These three strategies are described in Section II. The experimental results presented in Section IV have indicated that: the performance of CudaPre3D (i.e., the running time and speedups) varies when using different strategies. In this section, the effect of these strategies is analyzed in more details.



(a) For the points distributed in a cube



(b) For the points distributed in a sphere



(c) For the points derived from mesh models

Figure 5. Percentage of remaining points after discarding when using three variants of CudaPre3D

1) *Impact on the Running Time*

When selecting different strategies for rotating a set of input points, the number of rotating slightly differs. For example, it needs 9 times of rotating when using the Strategy 1, 7 times and 13 times when performing the Strategy 2 and Strategy 3, respectively. Obviously, the more the rotations are, the more computationally expensive it is. In other words, the running time spent on rotating 9 times is more than that of rotating 7 times. This leads to the results that Step 1 listed in Tables I ~ IX, cost different time when using different strategies of rotating.

After rotating and determining the extreme points, the second step of CudaPre3D is to form a convex polyhedron / hull for those determined extreme points. Due to the fact that the overhead for calculating the convex polyhedron is quite little (about 1ms in almost all cases, see Tables I ~ IX). Thus, the strategies of rotating nearly do not have any impact on the running time of this step.

But they obviously have the impact on the final shape of the convex polyhedron of the determined extreme points. This is due to the fact that: when using different strategies of rotating, typically different number of and positions of the extreme points can be found to be used to form the convex polyhedron. Thus the convex polyhedron that is in fact the convex hull of the found extreme points in general differs.

The final step of CudaPre3D is to discard those interior points. The key procedure in the discarding of interior points is to check that a point locate on the positive or negative side of a triangle facet of the convex polyhedron. Thus, the running time of discarding interior points heavily depends on the shape of the convex polyhedron. As analyzed above, the strategies of rotating affect the shape of the convex polyhedron. Therefore, the strategies of rotating also have an impact on the running time of discarding interior points.

2) Impact on the Effectiveness of Filtering

The effectiveness of filtering / discarding interior points is also highly affected by the shape of the convex polyhedron. This is because that: if the convex polyhedron formed by the determined extreme points is very close to the expected convex hull of an input set of points, then in general most input points can be determined to be located inside the convex polyhedron; and thus very few remaining points retain. In this case, the effectiveness of discarding interior points has a high possibility of being satisfied.

In summary, the strategies of rotating have the impact on the shape of the convex polyhedron formed by extreme points; and the effectiveness of discarding interior points heavily relies on the shape of the convex polyhedron. Thus, the strategies of rotating have a strong impact on the effectiveness of discarding interior points. This behavior has been illustrated in Fig.5.

C. Analyses of the Negative Results for the Points Derived from Mesh Models

The experimental results have showed that: the speedups in the cases where CudaPre3D is employed over the cases where CudaPre3D is not used are approximately 2x in most cases. These results also indicate that: the CudaPre3D is not applicable to those sets of points directly derived from 3D mesh models. Here we analyze the input data carefully, and try to explain why these negative results arise.

First, the CudaPre3D is strongly input-sensitive. That is, the performance of CudaPre3D such as its efficiency and effectiveness is strongly sensitive to the distribution of the input set of points. More specifically, (1) for the Step 1 of CudaPre3D, when using a specific strategy to rotate the input set of points, the number of and the positions of extreme points completely depends on the distribution of the input points; (2) for the Step 2, the shape of the convex polyhedron fully relies on the extreme points; (3) for the Step 3, the efficiency and effectiveness of discarding interior points are strongly affected by the shape of the convex polyhedron.

Second, those sets of input points derived from 3D mesh models distribute on the "surface / skin" of the 3D models. Take another case for example, if there is an volume mesh of a 3D mesh model containing lots of vertices and elements inside the mesh models, then the vertices of the original mesh model still lie on the "surface / skin", while there are lots of vertices of the volume elements such as tetrahedrons, prisms, or hexahedrons locating inside the mesh model. We term the first type of vertices as "surface / skin points", and the second type of vertices as "inner points".

Assuming that we use part of the vertices of the original mesh models, for example, the extreme points, to form a convex polyhedron / hull, and filter all the vertices by checking them whether they fall into the formed convex polyhedron, then those points laying on the "surface / skin", i.e., the surface points, typically have a high possibility of being determined outside, while those vertices locating inside the mesh models, i.e., the inner points, have a high possibility of being determined inside.

In an easy-to-understand manner, we can consider both the "surface points" and the "inner points" of a volume mesh models as a ***Dense*** point cloud; and there is no obvious "hole". Also, we can treat only the "surface points" as a ***Sparse*** point cloud which has an obvious "hole". When determining the interior points, it is obvious that there are much more interior points that can be found for the dense point cloud than that for the sparse point cloud. This also suggests that: the discarding of interior points for the dense point cloud is much more effective than that for the sparse point cloud.

Third, in this paper, we only use the points derived from 3D ***Surface*** mesh models, rather than 3D ***Volume*** mesh models. In this case, there are only "surface points" and no "inner points". Those "surface points" in general have a very high possibility of be determined to locate outside the convex polyhedron formed by some extreme points. Thus, the use of the convex polyhedron / hull to filter the input set of points is not effective.

D. Workload between Computations on the GPU and the CPU

For the CudaPre3D, both the first and the third steps are performed on the GPU, while the second step is carried out on the CPU. When employing the CudaPre3D and the Qhull to find the convex hull of an input set of points, the Qhull executes on the CPU.

The experimental results presented in Tables I ~ IX have indicated that: for those three groups of tests, the running time of the second step is approximately 1ms in most cases. This also suggests that almost all overhead when performing the CudaPre3D comes from the computations on the GPU rather than on the CPU. Furthermore, in most experimental tests, the third step, i.e., the discarding of interior points is more computationally expensive than the first step.

In addition, as analyzed in Section V, the efficiency and effectiveness of both the third step and the entire procedure of CudaPre3D are strongly input-sensitive. The workload between the first step and the third step also strongly relies on the distribution of the input sets of points. Moreover, the workload between the CudaPre3D and the Qhull heavily relies on the distribution of the input sets of points.

E. Correctness and Complexity

The correctness of the CudaPre3D can be obviously guaranteed. As introduced several times, the basic ideas behind CudaPre3D is to first find several groups of extreme points for the original input set of points and several rotated sets of the input points, and then discard those interior points that locate inside the convex polyhedron formed by the found extreme points.

The first idea is that: those input points with the min or max coordinates are definitely extreme points of the desired convex hull. And this statement does not need any proof. It is definitely correct.

The second idea is to find several groups of extreme points for the original input point set and the rotated version of the input point set. The idea of finding extreme points by rotating is due to the fact that: the extreme points of a convex hull for a specific set of points do not alter after all points are rotated in the same angle along the same direction. This step is obviously correct. In addition, the creating of a convex polyhedron formed by the found extreme points can also be correct using the incremental algorithm [20].

About the correctness of the discarding of interior points, According to the Grunbaum's Beneath-Beyond Theorem [21], all the points that locate inside the convex polyhedron cannot be the extreme points of the convex hull, and thus can be directly discarded. Therefore, the correctness of this step can also be guaranteed.

The time complexity of CudaPre3D is $O(n)$. The moving and rotating of a set of points, the determining of extreme points with min or max coordinates, and the discarding of interior points completely run in $O(n)$. And the forming of a convex polyhedron / hull using some previously determined extreme points only needs constant time. Thus, the entire algorithm runs in $O(n)$ time.

VI. CONCLUSION

We have presented a conceptually simple preprocessing algorithm, CudaPre3D, for accelerating the computation of convex hulls in three-dimensions on the GPU. The proposed algorithm, CudaPre3D, is an extension to our previous work CudaPre [19]. In CudaPre3D, we first find several groups of extreme points from the original set of input points and several rotated versions of the input point sets, then form a convex polyhedron / hull using the found extreme points, and finally discard those interior points that locate inside the formed convex polyhedron / hull. We have carried out three groups of experimental tests to evaluate the performance of CudaPre3D by comparing the efficiency when employing the proposed algorithm with the efficiency in the case where our algorithm was not adopted. The experimental results show that: the speedups in the case where CudaPre3D is adopted over the case where CudaPre3D is not employed are about 4x on average, and 5x ~ 6x in the best cases. In addition, when using CudaPre3D, more than 95% input points can be discarded in most tests. We have also observed that: for some types of test data, the CudaPre3D is not effective, and obtains unsatisfied speedups. But we believe that the CudaPre3D is an alternative choice in practice.

REFERENCES

[1] F. P. Preparata, M. I. Shamos, "Computational Geometry: An Introduction", pp. 131-136, New York: Springer-Verlag, 1985.

[2] F. P. Preparata, S. J. Hong, "Convex hulls of finite sets of points in two and three dimensions," Communications of the ACM, vol. 20, no. 2, pp. 87-93, 1977. [Online]. Available: http://dx.doi.org/10.1145/359423.359430

[3] K. L. Clarkson, P. W. Shor, "Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental," in Proceedings of the 4th symposium on Computational geometry, 1988, pp. 12-17. [Online]. Available: http://dx.doi.org/10.1145/73393.73395

[4] T. M. Chan, "Optimal output-sensitive convex hull algorithms in two and three dimensions," Discrete & Computational Geometry, vol. 16, no. 4, pp. 361-368, 1996. [Online]. Available: http://dx.doi.org/10.1007/BF02712873

[5] C. B. Barber, D. P. Dobkin, H. Huhdanpaa, "The quickhull algorithm for convex hulls," ACM Transactions on Mathematical Software (TOMS), vol. 22, no. 4, pp. 469-483, 1996. [Online]. Available: http://dx.doi.org/10.1145/235815.235821

[6] 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, pp. 1-5.

[7] S. Srungarapu, D. P. Reddy, K. Kothapalli, P. Narayanan, "Fast two dimensional convex hull on the GPU," in Advanced Information Networking and Applications (WAINA), 2011, pp. 7-12. [Online]. Available: http://dx.doi.org/ 10.1109/WAINA.2011.64

[8] A. Stein, E. Geva, J. El-Sana, "CudaHull: Fast parallel 3D convex hull on the GPU," Computers & Graphics, vol. 36, no. 4, pp. 265-271, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.cag.2012.02.012

[9] M. Tang, J.-y. Zhao, R.-f. Tong, D. Manocha, "GPU accelerated convex hull computation," Computers & Graphics, vol. 36, no. 5, pp. 498-506, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.cag.2012.03.015

[10] 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, San Francisco, California, 2011, pp. 204-204. [Online]. Available: http://dx.doi.org/ 10.1145/1944745.1944784

[11] M. Gao, T.-T. Cao, A. Nanjappa, T.-S. Tan, Z. Huang, "gHull: A GPU algorithm for 3D convex hull," ACM Transactions on Mathematical Software (TOMS), vol. 40, no. 1, pp. 1-19, 2013. [Online]. Available: http://dx.doi.org/10.1145/2513109.2513112

[12] M. Gao, T.-T. Cao, T.-S. Tan, Z. Huang, "Flip-flop: convex hull construction via star-shaped polyhedron in 3D," in Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2013, pp. 45-54. [Online]. Available: http://dx.doi.org/10.1145/2448196.2448203

[13] S. Tzeng, J. D. Owens, "Finding convex hulls using Quickhull on the GPU," arXiv:1201.2936, 2012.

[14] J. M. White, K. A. Wortman, "Divide-and-Conquer 3D convex hulls on the GPU," arXiv:1205.1171, 2012.

[15] B. Kim, K.-J. Kim, "Computing the convex hull for a set of spheres on a GPU," in Proceedings of the 11th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its Applications in Industry, Singapore, 2012, pp. 345-345. [Online]. Available: http://dx.doi.org/10.1145/2407516.2407596

[16] C. Xing, Z. Xiong, Y. Zhang, X. Wu, J. Dan, T. Zhang, "An eficient convex hull algorithm using affine transformation in planar point set," Arabian Journal for Science and Engineering, vol. 39, no. 11, pp. 7785-7793, 2014. [Online]. Available: http://dx.doi.org/10.1007/s13369-014-1365-3

[17] S. G. Akl, G. T. Toussaint, "A fast convex hull algorithm," Information Processing Letters, vol. 7, no. 5, pp. 219-222, 1978. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(78)90003-0

[18] J. Cadenas, G. Megson, "Rapid preconditioning of data for accelerating convex hull computations," Electronics Letters, vol. 50, no. 4, pp. 270-272, 2014. [Online]. Available: http://dx.doi.org/10.1049/el.2013.3507

[19] G. Mei, "A straightforward preprocessing approach for accelerating convex hull computations on the GPU," arXiv:1405.3454, 2014.

[20] M. Kallay, "The complexity of incremental convex hull algorithms in Rd," Information Processing Letters, vol. 19, no. 4, pp. 197, 1984. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(84)90084-X

[21] B. Grunbaum, "Measure of symmetry for convex sets, "In: Proceedings of the¨7th symposium in pure mathematics of the American mathematical society, 1961. pp. 233–70.