Parallel Surface Reconstruction on GPU

Heng Gao
State Key Laboratory for Novel Software Technology,
Department of Computer Science And Technology,
Nanjing University
Nanjing, China

Jie Tang*
State Key Laboratory for Novel Software Technology,
Department of Computer Science And Technology,
Nanjing University
Nanjing, China

Gangshan Wu
State Key Laboratory for Novel Software Technology,
Department of Computer Science And Technology,
Nanjing University
Nanjing, China

ABSTRACT

Marching Cubes is the most frequently used method to reconstruct isosurface from a point cloud. However, the point clouds are getting denser and denser, thus the efficiency of Marching cubes method has become an obstacle. This paper presents a novel GPU-based parallel surface reconstruction algorithm. The algorithm firstly creates a GPU-based uniform grid structure to manage point cloud. Then directed distances from vertices of cubes to the point cloud are computed in a newly put forwarded parallel way. Finally, after the generation of triangles, a space indexing scheme is adopted to reconstruct the connectivity of the resulted surface. The results show that our algorithm can run more than 10 times faster compared to the CPU-based implementations.

Categories and Subject Descriptors

I.1.3 [**COMPUTER GRAPHICS**]: Hardware Architecture - *Graphics processors, Parallel processing*.

General Terms

Algorithms, Performance.

Keywords

GPU, CUDA, Isosurface Extraction, Marching Cubes

1. INTRODUCTION

Isosurface extraction, usually from large scale scattered points, is widely used in scientific computing visualization, many scalar field visualization problems can be summed up as isosurface extraction and rendering， such as 3D reconstruction of medical images, presentation of molecular surface in molecular chemistry, structure analysis of mineral deposits distribution in geology and so on. Hoppe[1][2] achieved effective and pioneering work in the area.

The most widely used method of isosurface extraction is Marching Cubes (MC) algorithm [3]. MC algorithm has better rendering results in higher data density, but tends to generate too many triangles, which costs large storage space and transmission bandwidth and requires too much longer processing time, leading the real time rendering to be impossible. Although the cubes which intersect the isosurface in model space are only a small part of the total cubes, during the process of extracting isosurface, MC algorithm still needs to traverse all cube units to generate the surface patches, reducing the efficiency greatly. Furthermore, the storage of these cubic units which don't intersect the isosurface also wastes a large amount of valuable memory space.

There are two ways commonly used to improve the efficiency of the MC algorithm. The first one adopts octree [4] to organize and manage data, reduce the computing consumption of disjoint cubes and improve the method efficiency as well as save the memory. Another method is to execute the algorithm in a growing way, i.e. starting with preprocessing step to select a part of the cubes as seed cubes, and then use the regional search algorithm to traverse and find cubes intersect seed cubes, until all surface patches are found.

Nowadays, the scale of point clouds is getting larger and larger. Thus the algorithm still cannot meet the real-time requirements. Fortunately, with the development of modern programmable GPU, there appears a new solution for achieving large scale MC algorithm in real time or interactively. Löffler et al. [5] parallelize the dual marching cubes method [6] to extract manifold surfaces from terrain data set. Schmitz et al. [7] implement a modified dual contouring on the GPU. Chen [9] proposed a new parallel approach to efficiently construct high-quality polygon meshes from implicit surface representations.

In the paper, we present an isosurface extraction algorithm based on GPU. The algorithm first divides the model space into a uniform grid and distributes each sample point into a cube of the uniform grid. Then the directed distances from the vertices of each cube to the point cloud are calculated in a way suitable for SIMT (single instruction multi threads) parallel model. Finally, triangles are generated in all the cubes to form the isosurface in parallel. The experimental results show that the isosurface extraction algorithm in this paper can generate reconstructed surface efficiently.

2. Marching Cubes

Marching Cubes (MC) is a widely used isosurface extraction algorithm. Isosurface is a set in which all the elements share the same value of a function, it can be expressed as below,

$$\{\,(x, y, z) \mid f(x, y, z) = c\}, \text{ c is a constant value}$$

* corresponding author

MC algorithm divided the model space into cubes, assuming the data are continuously changing along the edges of the cube, which means: if the isosurface value is between two vertices of an edge, the isosurface will intersect the edge for sure. In order to determine the isosurface structure in a cube, a threshold has to be set at first. Then the 8 vertices of the cube are classified, depending on whether the vertex is inside or outside the isosurface. If the vertex value is greater than the threshold, then the vertex is set in the isosurface, denoted as "0"; if the vertex value is less than the threshold, then vertex is set outside of the isosurface, denoted as "1". Since each vertex has two states, each cube has a total of 2^8(256) combination states. From the topological point of view, the combination of 256 kinds of state will be reduced to 15 after the reversal and rotation transformation, as shown in Figure 1.

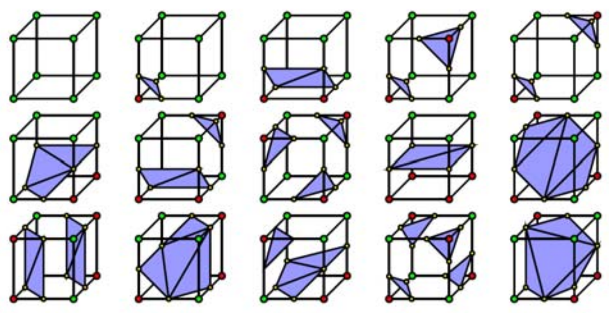
Figure 1. The 15 Basic Combinations of Cubes

MC algorithm checks each cube, determines its topological type, and looks for to which edge the cube vertices belong to, compute the intersected points by using the linear interpolation method, then form the corresponding triangles. After examining all possible triangles of the cubes, we can get the equivalent triangle meshes of the isosurface.

The algorithm of Hoppe [1] takes each sample point's tangent plane as the local linear approximation of the surface to be reconstructed, thus a directed distance function $f(\boldsymbol{c})$ can be constructed from a space point $\boldsymbol{c}$ to the interface to be reconstructed:

$$f(\mathbf{c}) = (\boldsymbol{c} - \boldsymbol{o_i})\, \boldsymbol{n_i}$$

Where $\boldsymbol{o_i}$ is the nearest tangent plane center to $\boldsymbol{c}$, and $\boldsymbol{n_i}$ is the the normal vector of the tangent plane.

The zero set of $f(\boldsymbol{c})$, which is denominated as Z(f), forms an isosurface. When we apply the MC algorithm to extract the triangles, the 256 different cube triangle structures are stored in a table at first. Then, the directed distance $f(\boldsymbol{c})$ between the eight vertices of each cube and the interface to be reconstructed are calculated, classifying the cube according to directed distances, a vertex with distance bigger than zero is labeled as "1", otherwise is labeled as "0". According to the data classification result of the cube, look up the related data in the table, use linear interpolation method to calculate the intersection point between the isosurface to be reconstructed and the cube, and finally output the corresponding triangle meshes.

3. Marching Cubes on GPU

For a large scale of point cloud which has over half million points, computing directed distance runs very slowly, which is caused by two reasons: first, the number of cubes which actually intersect isosurface are only part of the total cubes, most of the calculation of directed distance is invalid computation; second, to compute directed distance, a search for the nearest tangent plane center $\boldsymbol{o_i}$ to each vertex $\boldsymbol{c}$ is needed, which will traverse all the sampling points, but the sampling points are randomly distributed around each vertex, the distribution of each vertex is different. Therefore, this paper presents an efficient directed distance calculation method based on GPU.

3.1 Cube Indexing

MC algorithm divides the model space into small cubes, among which only a small part intersect the surfaces to be reconstructed, so how to quickly locate these cubes is the key to improve the efficiency of the algorithm. Assuming a point cloud X as the sampling points set of the surface M to be reconstructed, and the density is ρ, the noise is δ, the projection of a vertex $\boldsymbol{c}$ on the surface M is z. When $d(z, X)>\rho+\delta$, in which, $d(z, X)$ denotes the nearest distance between z and certain point in the point cloud X. the cube is considered to have no intersection with the surface. Also, according to the principles of MC algorithm, if all 8 vertices of a cube are on the same side of the surface, they won't intersect, either. From the above two properties, it's not hard to conclude, only in the cubes crossing the sampling points can isosurface triangles be generated.

Each cube has a three-dimensional subscript (x, y, z) in the model space to indicate its position in space. In order to save the storage space, the three-dimensional subscript of a cube is coded in a 32 bit unsigned integer, the first 10 bits denote X, the second 10 bits denote Y, and the third 10 bits denote Z. We call this number Cube Key, as shown in Figure 2. The maximum length of each dimension is 2^{10}, so the model space can be divided into 2^{30} (1G) cubes.

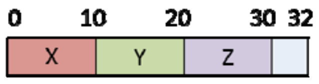

Figure 2. The Denotation of Cube Key

3.2 Locating Sample Points

In order to improve the efficiency of searching the sampling points, we take a strategy similar to Uniform Grid[7][12] to put sampling points into cubes. Different from those two methods, the cubes are not only used to improve the searching efficiency, but also to serve as the cubes in the MC algorithm to extract triangle meshes. When the length of edge for each cube is decided, the Cube Key of each cube to which every sampling point belongs can be computed parallel on GPU. As more than one sampling point falls into a cube, we use parallel primitive to sort [8] the sampling points by Cube Key, moving the points within the same cube together. Then we use parallel primitive compaction [10] to remove the duplicate Cube Key, the cubes left are called seed cube, preparing for the next region diffusion. A tag array is set to be the cut-off point, as shown in Figure 3. This series of actions are completed by the parallel primitives, so it has very high efficiency on GPU.

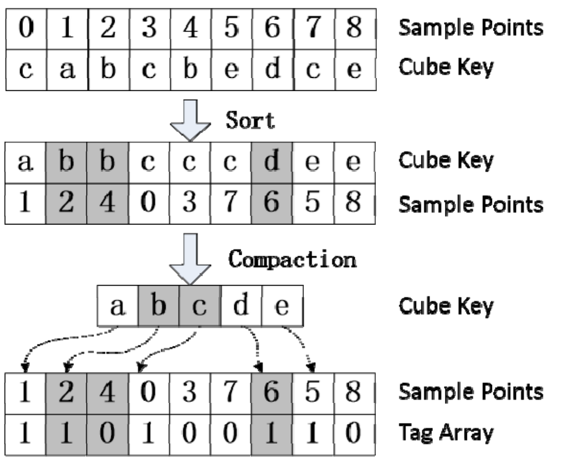

Figure 3. Allocating the sample points into each cube

3.3 Directed Distance

Directed distance is usually computed by sequential searching the nearest tangent plane center $\boldsymbol{o_i}$ of sampling points for each vertex $\boldsymbol{c}$. Since the sampling points are randomly distributed, the distribution of sampling sites around each vertex is different. CUDA is an SIMT (single instruction multi threads) parallel computing model, inconsistent behavior in parallel threads will produce branches, which will cause the execution paths running serially and reducing the program's parallel degree. On the other hand, reading random sampling data is not suitable for CUDA's combined global memory access mode.

To solve the problem, we present a new method to compute the directed distance for cube vertices. Instead of searching the nearest sampling points center $\boldsymbol{o_i}$ around cube vertex c, we compute the directed distance between tangent plane center $\boldsymbol{o_i}$ of each sampling point and 8 vertices of the cube it locates. Because of the determined position of vertices and their regular distribution, the parallel processing is more intuitive and efficient. With the cube, where sampling points are, being the center, a subspace is formed by spreading one layer out, which composes of 3^3 cubes and 4^4 cube vertices. Thus the model space is divided into a number of subspaces according to the seed cube's position. Figure 4. Shows a 2D explanation.

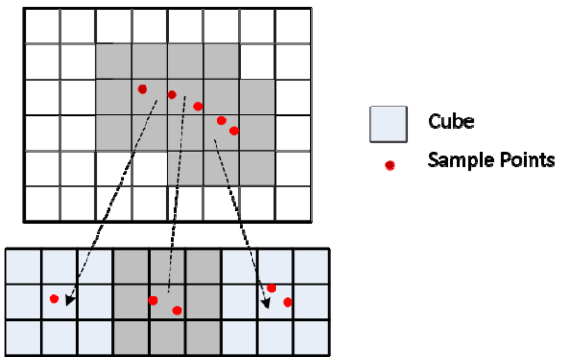

Figure 4. 2D Explanation of Dividing the Model Space

This algorithm maps each subspace into a block in CUDA, which means each block consists of 64 threads, and each thread processes one cube vertex. The sampling data is saved in the Shared Memory, shared by all thread in subspace, which greatly reduces the times of global memory access. Every subspace contains more than one sampling point, each cube vertex only calculates directed distance to the nearest sampling point center. In this way, the local directed distance within the subspace can be parallel obtained, the algorithm is as follows.

Algorithm 1 Algorithm to Compute the Directed Distance

```
1:  procedure __global__ void ComputeOriDist kernel()
2:      decode the Cube Key of seed cube, save in Shared Memory;
3:      get the subscript of sampling point in subspace, save in Shared Memory;
4:      synchronize the data in Shared Memory;
5:      compute the subscript of each vertex in subspace;
6:      for p ∈ Samplingpoint do
7:          get the tangent plane center and the normal vector of p, save in Shared Memory;
8:          synchronize the data in Shared Memory;
9:          compute the directed distance between a vertex and certain sampling point center;
10:         record the nearest sampling point info;
11:     end for;
12:     compute the directed distance function between the vertex and the sampling point;
13: end procedure
```

A vertex of cubes in the original model space may be mapped into more than one subspace, and be calculated many times, therefore an optimal value should be selected in the global scope, which is the nearest directed function between $\boldsymbol{c}$ and the sampling point center $\boldsymbol{o_i}$. For each cube vertex, similar to the Cube Key in Figure 2, we set a key called Vertex Key, and then sort the Vertex Keys by parallel primitives, thus the directed distance of the same cube will be put together, the directed distance corresponding to the nearest sampling point center is chosen to be the finally distance. At the same time, we use parallel primitives of sorting and compaction to remove duplicate cubes.

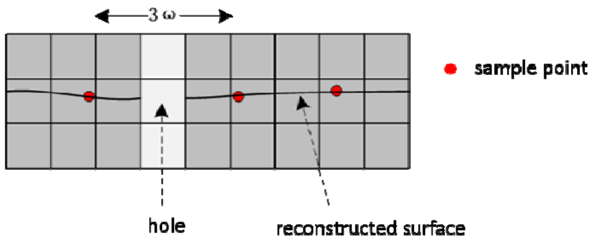

Figure 5. Density of sample points and hole

In the algorithm of this paper, sampling points are distributed into seed cubes. Only those key values of the first layer of neighborhood of the seed cubes (which shares the common vertex Cube) are retained, other key values will not be stored, thus we can both improve the computational efficiency and save the storage space. However, there is a risk of producing holes in the surface. Assuming that the cube edge length is ω, in the point cloud with density ρ, noise δ, if $3\omega<=\rho+\delta$, a surface may be reconstructed with holes, as shown in Figure 5, otherwise as long as $3\omega>\rho+\delta$, our method will not produce any holes in the reconstructed surface.

3.4 Marching Cubes

With the directed distance of vertex, the cubes can be classified by the combination of the signs of 8 directed distances. After looking up in the pre-set table by the cube classification, through linear interpolation, triangle meshes can be created inside a cub according to the information looked up in the table. As the treatment for each cube is independent from each other, the generation of triangles in cubes is performed on GPU.

The intersection points of cubes and reconstructed surface lie on the edges of cube. Their positions could be obtained by linear interpolation between the two vertex of the edge, the indices of vertices and edges are shown in Figure 6.

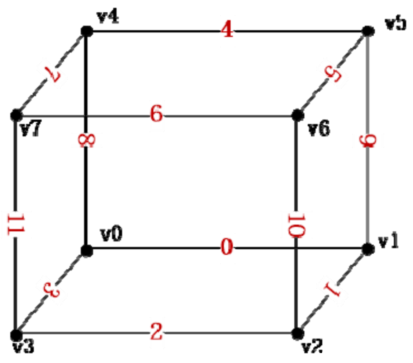

Figure 6. The indexing of Vertices and Edges

For example, in a certain cube, vertex 3 is inside the isosurface, and the other vertices are outside of the isosurface, as can be seen from Figure 6, the cube will generate a triangle, and its three vertices are located in edge 3, 11 and 2, therefore, a table called triTable is needed to record the edge set of triangle vertices for each classification of cubs. There are at most 5 triangles in one cube, as shown in Figure 1, so each vertex has up to 3*5=15 triangle vertex, which can be denoted by a 16 bytes array. For those with only 3 vertex, they can be donated as: {3,11,2, x, x, x, x, x, x, x, x, x, x, x, x, x}, in which x can be set to any number greater than the maximum number of edges, such as 255.

As there are 256 different cubes in total, a two-dimensional array of size 256*16 is allocated to represent the triTable, with records to the corresponding edge indices. In order to create the triangle in each cube in parallel, we also need to establish a numVertsTable to record the number of triangles in each kind of cubes, which will be used to determine the subscript of triangles in the output for each cube by primitives[11][13]. These two tables are both read-only and can be bounded to the GPU texture cache to improve the efficiency of the search.

Each cube is processed by one CUDA thread. The algorithm processes 12 edges of each cube one by one. In the parallel processing of every thread, the subscripts in the output array of generated triangles are needed to be known in advance, which means we need to classify the cubes and look up in numVertsTable to get the number of triangles. Then the subscript in the output array can be generated by scan primitives. The parallel generation of triangle algorithm is shown as below:

Algorithm 2 Algorithm to Generate Triangle

```
1:  procedure __global__ void generateTriangles kernel()
2:      decode the number of cube vertex, get the coordinates of vertices;
3:      get the directed distance of cube vertex;
4:      traverse the vertices of the cube, determine the classification of the cube;
5:      perform linear interpolation for the 12 edges one by one to get the intersection;
6:      look up in numVertsTable, get the number of triangles;
7:      for all triangles do
8:          look up in the triTable, get the number of edge on which triangle vertices are;
9:          according to be index of edge, get the intersection point as the triangle vertex;
10:         save the triangle vertex to the output array, the subscript is computed in advance;
11:     end for;
12: end procedure
```

The reconstructed surface is composed of the triangles generated in all cubes. The output we need is an array of the vertices of all triangles. Every three continuous represents a triangle. As each cube generates its own triangle vertex, there will be duplicate triangle vertex. It's necessary to remove the duplicate vertex, and update the vertex index for each triangle. The duplicate vertices are removed by parallel primitives of sort and compaction. And the vertex indices of triangles are updated with parallel primitives of sort and scan.

4. RESULTS

The proposed isosurface extraction algorithm based on GPU is implemented on CUDA 7 with Thrust 1.7.0 parallel primitives library. The experimental platform is: Windows 7, CPU Intel Core (TM) 2 Duo E6550 (dual core), 4GB memory, GPU NVIDIA GeForce GTX 660, the CUDA stream processor number is 192, the processor frequency is 1.57GHz, the memory is 1GB.

Figure 7 and Figure 8 show the graphical results of our proposed isosurface extraction algorithm working on different point clouds. For large scale point clouds, this algorithm can still quickly reconstruct a fine mesh surface. The experimental data of Dragon and Buddha point clouds are from the Stanford University 3D database. We compare the running time of GPU algorithm and CPU algorithm. The CPU version algorithm has been optimized, Detailed data are shown in Table 1.

Table 1. Time Results of Isosurface Extraction

Model	#Points	# Output Triangles	GPU	CPU	SP*
Dragon	437645	244325	10.348s	112.461	11
Buddha	543625	331225	11.094s	146.023	13

*Speed Up

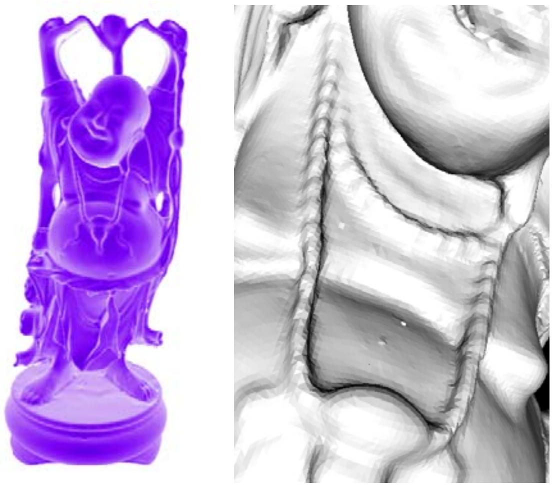

Figure 7. Graphical results of reconstructed surface of Budha. The left one is a global view. The right one shows a detailed area.

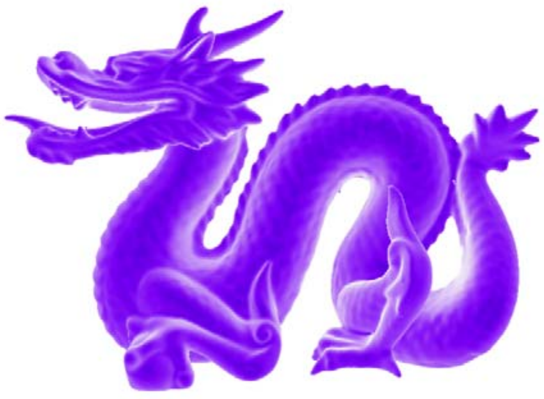

Figure 8. The reconstructed surface of Dragon.

From the results shown in Table 1, we could find that our isosurface extraction algorithm based on GPU runs about 10 times faster than CPU version does. With the increasing of the number of triangles, the speedup gets more obvious. The speedups of complex point clouds such as Dragon, Buddha are better than that of relatively simple structure such as a fat surface.

5. CONCLUSION

Isosurface extraction is an important technique widely used in the field of visualization in scientific computing. However, the point clouds are getting denser and denser, thus the efficiency of Marching cubes method has become an obstacle. This paper presents a novel GPU-based parallel surface reconstruction algorithm. The algorithm firstly creates a GPU-based uniform grid structure to manage point cloud. Then directed distances from vertices of cubes to the point cloud are computed in a novel parallel way. Finally, after the generation of triangles, a space indexing scheme is adopted to output the resulted surface in a ply like format. The results show that our algorithm can run more than 10 times faster compared to the CPU-based implementations.

The memory of a GPU is limited which makes it difficult for GPU-based reconstruction methods to process extreme large scale point cloud. The future work includes streaming reconstruction on GPU which could fulfill the reconstruction and data transferring between RAM and GPU at the same time.

ACKNOWLEDGMENTS

This paper is partly funded by National Science and Technology Major Project of the Ministry of Science and Technology of China under grant 2011ZX05035-004-004HZ.

REFERENCES

[1] Hoppe H., DeRose T., et al. 1992. Surface reconstruction from unorganized points. Proc. ACM SIGGRAPH’92, 71-78.

[2] Hoppe H., DeRose T., et al. 1994. Piecewise smooth surface reconstruction. Proc. ACM SIGGRAPH'94, 295-302.

[3] Lorensen W. E. and Cline H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. Computer Graphics, 21(4):163-169.

[4] Wilhelms J. and Gelder A. V. 2000. Octrees for faster isosurface generation. IEEE Transactions on Medical Imaging, 19:739-758.

[5] Löffler, F., Schumann, H. 2012. Generating smooth high-quality isosurfaces for interactive modeling and visualization of complex terrains. In: Proceedings of the Vision, Modeling, and Visualization Workshop

[6] Nielson, M. 2004. Dual marching cubes. IEEE Visualization, 489–496.

[7] Schmitz, A., Dietrich, A., Comba, D. 2009. Efficient and high quality contouring of isosurfaces on uniform grids. In: IEEE XXII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI), 64–71.

[8] Satish N., Harris M. and Garland M, 2009. Designing efficient sorting algorithms for manycore gpus. Parallel and Distributed Processing Symposium, pages 1-10.

[9] Chen J., Jin X., Deng Z. 2015. GPU-based polygonization and optimization for implicit surfaces. Vis Comput, 31:119–130..

[10] Nvidia. Cudpp:cuda data-parallel primitives library. http://www.gpgpu.org/developer/cudpp/, 2015.

[11] Sengupta S., Harris M. and Zhang Y., et al. 2007. Scan primitives for gpu computing. Graphics Hardware 2007, 97-106.

[12] Tang J., and Zhang F. 2005. Evaluation of similarity between arbitrary meshes. Journal of System Simulation, 17:16-19 (in Chinese).

[13] Dotsenko Y., Govindaraju N., et al. 2008. Fast scan algorithms on graphics processors. In Proceedings of the 22nd Annual International Conference on Supercomputing. 205-213.