2017 13th IEEE Conference on Automation Science and Engineering (CASE)
Xi'an, China, August 20-23, 2017

A GPU-based Parallel Slicer for 3D Printing

Xipeng Zhang[1], Gang Xiong[2], Zhen Shen[3], *Member IEEE*, Yiyao Zhao[4], Chao Guo[5], Xisong Dong[6]

***Abstract*— Mesh slicing is one of the most common operations in additive manufacturing (AM). However, the computing burden for such an application is usually very heavy, especially when dealing with large models. Nowadays the graphics processing units (GPU) have abundant resources and it is reasonable to utilize the computing power of GPU for mesh slicing. In the paper, we propose a parallel implementation of the slicing algorithm using GPU. We test the GPU-accelerated slicer on several models and obtain a speedup factor of about 30 when dealing with large models, compared with the CPU implementation. Results show the power of GPU on the mesh slicing problem. In the future, we will extend our work and standardize the slicing process.**

I. INTRODUCTION

Driven by the growing tendency of democratized manufacturing, three-dimensional (3D) printing has drawn much attention in the past few years. It has emerged as an important technology for manufacturing and found use in a wide range of fields. Examples of these include biomedical engineering [1], aerospace [2], medicine [3], military [4], automobile [5] and architecture [6]. Compared with traditional factory production, 3D printing has the advantage of less energy consuming and more economic benefits. The futurologist Jeremy Rifkin claims that the 3D printing will succeed the tradition production line assembly manufacturing mode and open a door to the third industrial revolution [7].

3D printing, also known as additive manufacturing (AM), refers to processes of building solid objects by adding material layer by layer. Every 3D print starts from a 3D design file, a digital representation for a physical object. Though until now lots of 3D printing technologies, such as fused deposition modeling (FDM), selective laser sintering (SLS), and large area maskless photopolymerization (LAMP), have been developed, slicing the design file into thin layers is an essential step in all AM processes. The accuracy of slicing results has a direct impact on the final surface quality. Since the slicing process depends on 3D files, we would like to introduce volumetric representations firstly.

Volumetric representations provide foundations for a wide range of fields including computational geometry, computer-aided design (CAD), visualization, and virtual reality. In computer graphics, 3D objects are viewed as a collection of surfaces, where polygons play a dominant role in approximating arbitrary surfaces by meshes [8].

Slicing a 3D object refers to processes used to calculate the intersection of slice plane and these polygons and obtain layered machining paths in 3D printing [9]. It not only provides visual layered image representation of the volume data, but also enables advanced imaging algorithms for further analysis. As a typical application of reducing three-dimensional models to 2.5 dimensional layers, it has been integrated in many commercial software packages, such as SolidWorks [10], Autodesk 123D [11], Materialise Magics [12], MeshLab [13], etc. The volumetric representations and slicing process are illustrated in Figure 1.

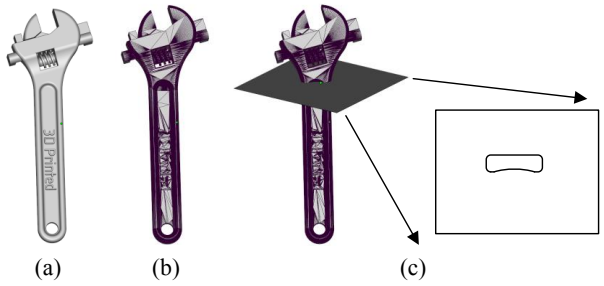

Fig. 1. Volumetric representations and slicing process: (a) a CAD model, (b) a tessellated model, (c) mesh slicing and contour generation

Mesh slicing is a well studied problem in the literature and various slicing algorithms have been proposed to optimize this engineering issue from the perspective of time complexity, memory usage, or model quality [14]. However, even the most time-efficient algorithm is far from satisfactory when dealing with large or high-precision models. Furthermore, the slicing process involves lots of independent and computation-intensive operations and intrinsically fits into parallel implementation. Employing parallel implementation and multi-core device can help to greatly speedup the process, especially for real-time system.

Compared with computer cluster, graphics processing unit (GPU) is an efficient and cost effective option for parallel computing. And it has become an integral part of today's common computer. In recent years, GPU technology has gone through a revolution and its application is far beyond graphics rendering. The integration of General-Proposed

[1]Xipeng Zhang is with the School of Computer and Control Engineering, University of Chinese Academy of Sciences, Beijing 100049, China. zhangxipeng2015@ia.ac.cn

[2]Gang Xiong is with Cloud Computing Center, Chinese Academy of Sciences, Dongguan 523808, China. gang.xiong@ia.ac.cn

[3]Zhen Shen is the correspondence author. He is affiliated with the Beijing Engineering Research Center of Intelligent Systems and Technology, Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China. zhen.shen@ia.ac.cn

[4]Yiyao Zhao with School of the Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China. zhaoyiyao506@stu.xjtu.edu.cn

[5]Chao Guo with the State Key Laboratory of Management and Control for Complex Systems, Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China. guochao2014@ia.ac.cn

[6]Xisong Dong with the Qingdao Academy of Intelligent Industries, Qingdao, Shandong 266109, China. xisong.dong@ia.ac.cn

GPU (GPGPU), heterogeneous computing and software development kits (SDKs) enables GPU to find wide use in scientific computing. Examples of these include fluid dynamic simulation [15], traffic engineering [16], bioinformatics [17] and geophysics [18]. A detailed introduction about GPU and its framework can be found in Section II.

In the paper, we propose a parallel mesh slicing implementation and develop a fast 3D slicer with the help of GPU. Our aim here is to study the potential and possible drawbacks of GPU in the application.

The rest of the paper is organized as follows. Section II gives a review on mesh slicing algorithm and GPU. A brief explanation of asymptotically optimal mesh slicing algorithm is presented in Section III. Section IV deals with the parallel implementation as well as two key techniques. Section V represents experimental results and the paper is ended in Section VI with conclusion and future work.

II. REVIEW

A. Mesh Slicing Algorithm

There are two variants of the slicing problem used in additive manufacturing. One is adaptive slicing which allows slices of variable thickness. The main interest of adaptive slicing is to make use of related information about the model to adjust the layer thickness. (See [19] for a review). The other assumes constant thickness in most cases. Research on this variant focuses on generating object contour for each slice as efficiently as possible [20]. In the paper, we deal with uniform thickness slicing problem.

Slicing tremendous meshes could cause heavy computing burden and large memory overhead, especially for complex models. Most slicing strategies in the literature concern with one optimization index at the expense the other. Vatani *et al*. proposed a slicing algorithm to optimize memory usage by only storing facets that intersect with current cutting plane [21]. Choi *et al*. proposed a tolerant slicing algorithm, which can effectively repair resultant inconsistent contours. They used the strategy of extracting one facet for slicing at a time to minimize memory usage [22]. These techniques are ideal for a small memory system but increase extra CPU overhead or time cost.

Instead of performing independent slice calculation, McMains *et al*. exploited coherence between consecutive slices and designed a coherent sweep plane slicer. The proposed topological data structure has an advantage of allocating memory more efficiently [23]. Huang *et al*. established a hash table to store coordinates of slicing planes so that all slicing planes that intersect with a given facet can be quickly found [24]. These techniques help to reduce slicing time but are difficult to parallelize.

Some parallel computing solutions for mesh slicing problem are found in the literature [14], [20], [25]. Given the tradeoff between memory usage and time efficiency, we choose the asymptotically optimal slicing algorithm and implement it with GPU.

B. NVIDIA GPU and CUDA Framework

As a multi-core device, GPU is specialized for compute-intensive and highly parallel tasks, which is the major difference from CPU. The latest NVIDIA GTX 1080 Ti GPU has up to 3584 cores with a theoretical memory bandwidth of 484GB/s. The cores are called streaming processors (SP) and several cores (8 or 32 typically) are organized into a streaming multi-processor (SM) [16]. The abundant resources in GPU make it most suitable for single instruction, multiple data (SIMD) operations, which bear the most computing burden in the slicing problem.

In November 2006, NVIDIA company released Compute Unified Device Architecture, also named CUDA, a general purpose computing platform and programming model [26]. With CUDA, people can have easy access to raw computing power in GPU and program GPU with high-level languages such as C, FORTRAN and OpenACC. A typical CUDA program consists of two parts. One is CPU (host) codes that handle sequential work, allocate and free GPU memory, transfer data between host memory and device memory and assign tasks to GPU. The other is GPU (device) codes that do parallel work. Functions with decorators *global* are so called "kernels". These kernels execute on GPU and can be called from the host.

Once a kernel is launched, multiple threads that run simultaneously will be activated. In CUDA, these threads are organized into a two-level hierarchy: block and grid. A block is made up of a three-dimensional array of threads, and a grid is made up of a three-dimensional array of blocks. The threads in the same block enjoy an additional shared memory and can be synchronized with each other. All threads can exchange data with the constant memory, texture memory and global memory of GPU. In addition, constant memory is a small and cached space for constant variables, and texture memory is optimized for 2D spatial locality. Global memory occupies the largest space of the GPU memory but has the largest latency. The CUDA framework tries to mask the latency by computing other threads. Fig. 2 illustrates the CUDA programming model.

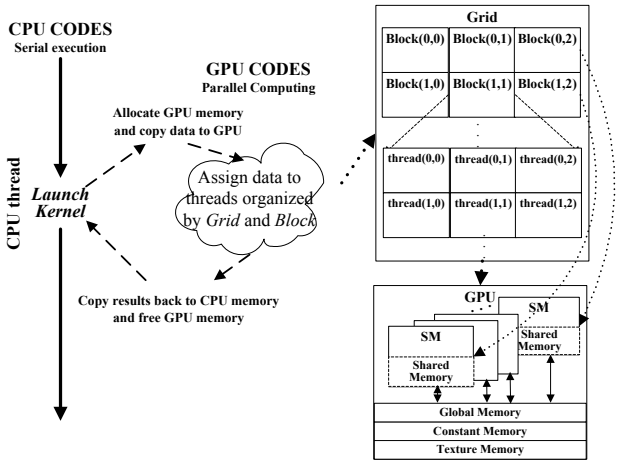

Fig. 2. An overview of the CUDA programming model

III. ALGORITHM

Before describing the method, we make some restrictions and give the formulation of the slicing problem.

- Triangles are the most commonly used polygons in representing 3D object, and the *de facto* industry format for storing it is the STL (StereoLithography) file. The format is a tessellated representation of a solid model, whose surface is approximated by a number of three sided planar facets (triangles). Since the STL file format has been the most popular for additive manufacturing, we here focus on the STL model slicing problem [27]. Note that in the new Additive Manufacturing File Format (AMF) standard, triangles are still used as the basic elements to define the volume information of a 3D object [28]. So, our method is also applicable to the next generation AM file format, also known as STL2.
- Rotation sometimes needs to be applied to the model before slicing and this may lead to different slicing results compared with the untransformed model. A model transformed with rotation will be treated as a new model with the coordinates of its points renewed by a rotation matrix.

The volumetric model *T* in a STL file is actually a set of triangles

$$T = \{t_i,\ i = 1, 2, \ldots, n\}$$

$$t_i = \{(V_0, V_1, V_2) \mid V_0, V_1, V_2 \in \mathbf{R}^3\}$$

where t_i is a triangle and V_0, V_1, V_2 represent the three vertices of t_i using a three-dimensional Cartesian coordinate system. By default, the slice plane is perpendicular to z axis. Assume that z_{min} and z_{max} are the minimum and maximum z coordinates of *T* and $slice\_thickness$ is the thickness of the slices. Thus, the number of slices *m* can be calculated by

$$m = \lceil \frac{z\_max - z\_min}{slice\_thickness} \rceil$$

where the ceiling function $\lceil \cdot \rceil$ is used to make *m* a integer.

In [20], Dant proposed the asymptotically optimal mesh slicing algorithm by three primary algorithms: preprocessing sort algorithm, slice algorithm and contour assembly algorithm. In the preprocessing sort algorithm, triangles are grouped into different sets, and then the intersections of slice planes and these triangles are calculated by the slice algorithm. The output of the slice algorithm is a collection of unsorted line segments, and the contour assembly algorithm creates complete closed contours for each layer. Since the slice algorithm is necessary for all AM technologies and consumes the most execution time, we here focus on the parallel implementation of the slice algorithm. To help you have a better understanding of our study, we give a brief explanation of the first two primary algorithms.

A. Preprocessing Sort Algorithm

For a given triangle, the slice planes intersecting with it can be easily calculated according to its minimum and maximum z coordinates. The idea behind precessing sort algorithm is to group together triangles so that all triangles that intersect with the same plane would be stored in the same bucket. Algorithm 1 shows the preprocessing sort algorithm. In most cases, the number of buckets in the bucket list *B* is equal to the number of slices m. For n triangles in *T*, the time complexity of this approach is $\mathcal{O}(n)$ [20]. Note that the triangles that is parallel to slice plane will be dropped since $Index_{highest}$ is equal to $Index_{lowest}$.

Algorithm 1: Preprocessing Sort

Input: Triangles T
Output: Bucket List B

$B \leftarrow$ Create a list of buckets();
foreach $triangle\ t\ in\ T$ **do**
  Find min and max z axis coordinate for t;
  Let $Index_{lowest}$ be the index of the lowest bucket intersecting t;
  $Index_{lowest}$=$\lceil \frac{triangle_{min_z} - z_{min}}{slice\_thickness} \rceil$;
  Similarly we have $Index_{higtest}$=$\lceil \frac{triangle_{max_z} - z_{min}}{slice\_thickness} \rceil$;
  for i = $Index_{lowest}$ *to* $Index_{higtest} - 1$ **do**
    $B[i].Add(t)$;
return B;

B. Slice Algorithm

With sorted triangles, slice algorithm uses a nested loop to calculate such a intersection line for every triangle in each bucket. The outer loop iterates over every bucket in the bucket list *B* and the inner loop iterates over every triangle in current bucket. Algorithm 2 shows the slice algorithm. The time complexity of slice algorithm can be simplified as $\mathcal{O}(n)$ [20].

Algorithm 2: Slice

Input: Bucket List B
Output: Line Segments S

$S \leftarrow$ Create a list of lines();
i=0;
foreach $bucket\ b\ in\ B$ **do**
  foreach $triangle\ t\ in\ b$ **do**
    if $B.SlicePlane \cap t \neq \emptyset$ **then**
      $line$=$Calculate\_Intersection$();
      $S[i].Add(line)$;
  i++;
return S;

According to the results of preprocessing algorithm, position relation between a triangle and a given slice plane can be covered by any of the five cases, shown in Fig 3. When only the highest vertex or the highest two vertices of a triangle are on the slice plane (case 1 or case 2), there is no intersection line. In other cases, there is a line created by the intersection of a slice plane and a triangle. In case 3,

one point of intersection is viewed as a line segment with the same end point. The coordinates of these intersections can be calculated by similar triangles and the process, in Algorithm 2, is represented by $Calculate\_Intersection()$ for simplicity.

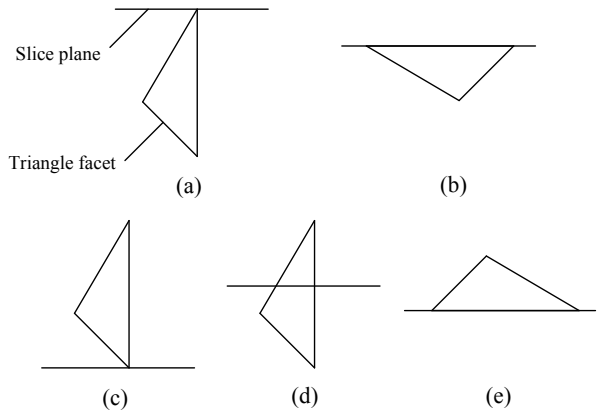

Fig. 3. All 5 cases when a slice plane intersects a triangle facet.

IV. IMPLEMENTATION

In this section, we mainly describe the parallel implementation of the slice algorithm. The basic idea behind our implementation is that every thread only deals with the calculation of the intersection of one triangle and one slice plane. In this way, we can maximize the parallel implementation of the nested loop in the slice algorithm. In the following description, two key techniques, data structure and computing resource allocation, in our implementation will be discussed.

A. Data Structure

Before slicing, we need to parse the 3D data stored in a STL file. The STL format specifies ASCII and binary representations and both representations contain an unordered list of triangles. Each triangle is define by the unit normal and vertices by 12 floating point numbers. By default, the direction of the normal and the order of vertices follow the right-hand rule.

Given the memory usage, we design two classes, *Vertex* and *Facet*, to store 3D data in a STL file. *Vertex* class has three floating numbers which represent the *X*/*Y*/*Z* coordinate of a vertex respectively. All different vertices of a STL model are placed in an array of *Vertex* and the index of these vertices depends on the order they appear. *Facet* class defines three integer variables to indicate the index of three vertices in a triangle mesh. Similarly, all triangles in a STL file are placed in a array of *Facet*.

Although the size of the bucket list *B* is unknown during the execution of preprocessing, we use a data structure of fixed-size array to implement *B*. The array size is obtained by adding up the number of triangles in each bucket. The triangles in *B* are represented by their index in the array of *Facet*. In this way, we can get access to all necessary data for slicing with the benefit of less memory usage.

B. Computing Resources Allocation

After preprocessing, large host variables, including bucket list *B* and STL model data (array of *Vertex* and array of *Facet*), are copied to the global memory of GPU, while some small variables, including $slice\_thickness$, m, and z_{min}, are copied to the constant memory for fast access by each thread.

In our high-level parallel implementation of the slice algorithm, every triangle in a bucket is assigned to one thread and all buckets will be executed simultaneously. In order to set aside enough threads for each bucket, an alternative upper limit of the total number of threads can be calculated as:

$$TotalThreadNum = m \times MaxBuctetSize$$

where m is the number of buckets in *B* and $MaxBucketSize$ is the maximum bucket size:

$$MaxBucketSize = \max\{\ |B[i]|\ \}$$

where $|B[i]|$ represents the size of bucket $B[i]$.

As shown in Fig. 4, a one-dimension block of threads $BlockDim$ and a two dimensions grid of thread blocks $GridDim$ are set up to organize these threads in GPU. There is a limit to the number of threads per block and optimal $BlockDim$ can be determined by several experiments or CUDA Occupancy Calculator. CUDA Occupancy calculator is an Excel spreadsheet that allows you to calculate the multiprocessor occupancy of a GPU by a given kernel. Note that the higher CUDA occupancy is, the more effectively you use the GPU. For more details, please see [29].

Given that $MaxBucketSize$ is usually much greater than $BlockDim$, several blocks are need to handle the triangles in a bucket. This forms the second dimension of $GridDim$:

$$GridDim.y = \lceil MaxBucketSize/BlockDim \rceil$$

where $GridDim.y$ represents the size of the second dimension of $GridDim$.

Each row of blocks in the grid handles the triangles in one bucket and the number of rows:

$$GridDim.x = m$$

where $GridDim.x$ similarly represents the size of the first dimension of $GridDim$.

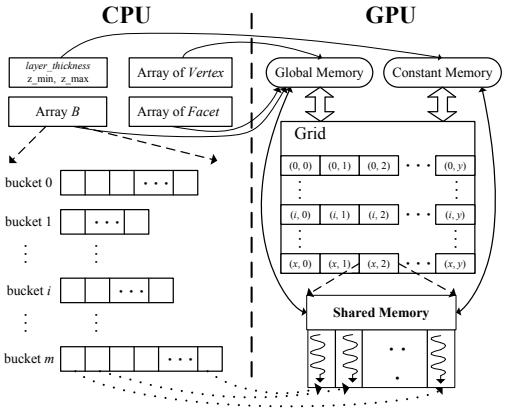

Fig. 4. Data structure and computing resource allocations

V. EXPERIMENT

A. Hardware Setup

The system used in our study contains a hexa-core Intel Xeon® E5-2620 at 2.0GHz with 128GB of DDR3 memory and a NVIDIA TITAN X card. Some key performance indicators of the GPU are given in Table I. For more technical data about the GPU, please refer to [30]. The machine runs a GUN/Linux system with Linux kernel 2.6 and NVIDIA driver 367.57. We use GCC 4.8.2 and NVIDIA CUDA 8.0 compiler for GPU code.

TABLE I
KEY PERFORMANCE INDICATORS OF NVIDIA TITAN X

Architecture	Maxwell
Number of SPs (cores)	3072
Clock Rate (*MHz*)	1076
Memory Bandwidth (*GB/s*)	336.5
Video Memory (*MB*)	12288

We deal with the kernel thread configuration issue at first. The number of threads per block, $BlockDim$, should be a round multiple of the warp size, usually 32 on most current hardware. Give that the maximum number of threads per block in TITAN X is 1024, some performance tests are made to find that the optimal $BlockDim$, in our case, is 1024. The grid size $GirdDim$ will be determined at run time using the equations provided in Part *B* Section IV.

B. Results

We implement the fast slicer as a console application for possible extension and further integration. Our aim here is to obtain slice results as fast as possible, so no graphical interface is desired.

To evaluate the performance of the fast slicer, we slice models of various size and measure the time spent on the parallel slice algorithm. Note that all test models can be obtained for free form the online community Pinshape [31]. We make a CPU implementation of the slicer as benchmark in our test. After 20 times independent run, a comparison of the average execution time for both implementations is shown in Table II.

TABLE II
TIME CONSUMPTION OF SLICING ON DIFFERENT MODELS

Model	# of Vertices	# of Facets	thickness	CPU Only (/s)	CPU+ GPU (/s)	Speedup
Elephant	62k	125k	1	0.36	0.23	1.57
Bracelet_c	121k	244k	0.5	1.62	0.30	5.40
Bracelet_d	243k	488k	0.1	10.56	0.56	18.86
Yoda	667k	1335k	0.05	29.59	1.35	21.92
Turbine engine	1527k	3059k	0.05	128.69	4.37	29.45
Clementine	2318k	4644k	0.05	239.72	7.03	34.10

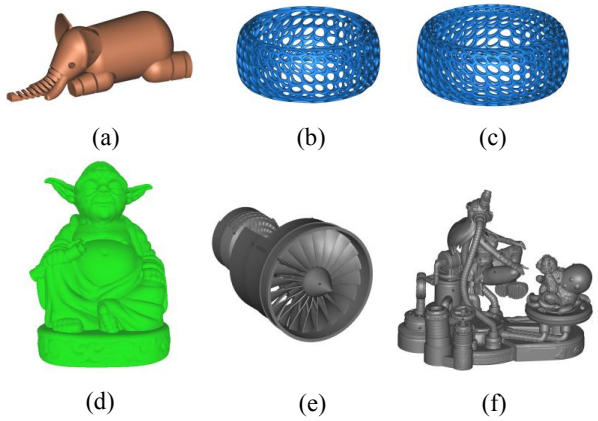

Fig. 5. Test models: (a) Elephant, (b) Bracelet_c, (c) Bracelet_d, (d) Yoda, (e) Turbine engine, (f) Clementine

As shown in Table II, the GPU-accelerated slicer gains decent speedups in all test models. Furthermore, along with the models become more complicated and the slice thickness becomes thinner, the computing burden is much heavier than before and it is time that GPU shows its power in the meshing slicing problem. We can conclude that parallel slicer has some advantages over tradition slicer and it can gain a speedup factor of about 30 when faced with complex models.

We use the CUPTI profiling library provided by CUDA to analyse the execution time of the parallel slice algorithm and find that the execution time is composed of two parts: data processing time and kernel execution time. As illustrated in Fig. 2, we allocate GPU memory and then transfer data to GPU before the kernel is launched. In our case, the data is STL data (array of *Vertex* and array of *Facet*), array *B* and some small variables. The size of array *B* makes up over 90% of the data and the percentage will continue to increase when the models become more complicated or the slice thickness becomes thinner. Array *B* can also affect the execution time of the kernel since it is closely related to thread configuration of the kernel. Therefore, we can reduce that array *B* has a significant impact on the execution time of the parallel slice algorithm.

We slice models with different thickness to make a approximately linear growth of the size of array *B*. The execution time for both implementations with two models is summarized in Table III and Table IV, which is also the average of 20 times independent run.

TABLE III
TIME CONSUMPTION OF SLICING ON ELEPHANT

Model	thickness	Size of Array *B* (/MB)	CPU Only (/s)	CPU+ GPU (/s)	Speedup
Elephant	1	20.3	0.36	0.23	1.57
	0.1	203.0	3.62	0.36	10.06
	0.01	2029.8	36.16	1.46	24.77

TABLE VI

TIME CONSUMPTION OF SLICING ON BRACELET_D

Model	thickness	Size of Array B (MB)	CPU Only (/s)	CPU+ GPU (/s)	Speedup
Bracelet_d	1	55.5	1.06	0.27	3.92
	0.1	554.7	10.56	0.56	18.86
	0.01	5547.3	104.93	3.34	31.42

From Table III and Table IV, it could be realized that the computation load of the slice algorithm has a nearly linear increase with the decrease the slice thickness, which is reflected in the execution time of the CPU implementation. However, the execution time of the CPU+GPU implementation grows rather slowly when the slice thickness decreases linearly and the growth in the speedup is a good indicator to show the potential of GPU in the slicing issues.

VI. CONCLUSION

In the paper, we propose a parallel implementation for the asymptotically optimal mesh slicing algorithm. GPU is used in our implementation and some details about data structure and computing resources allocation are given. Results show the great power of GPU on the mesh slicing issues.

The paper is part results of the on-going research project. Further research on contour assembly and the introduction of adaptive slicing is needed to fulfill the project. We believe that the standard procedure for parallel slicing can reasonably save slicing time, fully utilize current hardware resource and contribute to the development of 3D printing.

ACKNOWLEDGMENT

The work in this paper is supported by the National Natural Science Foundation of China under Grants 61233001, 71232006, 61304201, 61533019 and 91520301; 2016 College Students Innovation and Practice Training Program, Chinese Academy of Sciences; Finnish TEKES's Project SoMa2020: Social Manufacturing (2015-2017, 211560); Chinese Guangdong's S&T project (2014B010118001, 2014B090902001, 2014A050503004, 2015B010103001, 2016B090910001).

REFERENCES

[1] F. Rengier, A. Mehndiratta, and H. von Tengg-Kobligk *et. al*, "3D printing based on imaging data: review of medical applications," *International journal of computer assisted radiology and surgery*, vol. 5, no. 4, pp. 335–341, 2010.

[2] D. Bak, "Rapid prototyping or rapid production? 3d printing processes move industry towards the latter," *Assembly Automation*, vol. 23, no. 4, pp. 340–345, 2003.

[3] S. Chen, Z. Pan, Y. Wu, *et. al*, "The role of three-dimensional printed models of skull in anatomy education: a randomized controlled trail," *Scientific Reports*, vol. 7, no. 1, p. 575, 2017.

[4] C. C. Kai, "Three-dimensional rapid prototyping technologies and key development areas," *Computing & Control Engineering Journal*, vol. 5, no. 4, pp. 200–206, 1994.

[5] N. J. Mankovich, A. M. Cheeseman, and N. G. Stoker, "The display of three-dimensional anatomy with stereolithographic models," *Journal of digital imaging*, vol. 3, no. 3, pp. 200–203, 1990.

[6] C. X. F. Lam and X. M. Moand and S. H. Teoh, *et. al*, "Scaffold development using 3d printing with a starch-based polymer," *Materials Science and Engineering: C*, vol. 20, no. 1, pp. 49–56, 2002.

[7] (2016) Jeremy Rifkin and The Third Industrial Revolution Home Page. [Online]. Available: http://www.thethirdindustrialrevolution.com/

[8] H. Hsieh, Y. Lai, and W. T. *et al.*, "A flexible 3D slicer for voxelization using graphics hardware," in *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, Dunedin, New Zealand, 2005, p. 285.

[9] A. Dolenc and I. Mäkelä, "Slicing procedures for layered manufacturing techniques," *Computer-Aided Design*, vol. 26, no. 2, pp. 119–126, 1994.

[10] (2014) Solidworks. [Online]. Available: http://www.solidworks.com.cn/

[11] (2010) Autodesk 123d. [Online]. Available: http://www.123dapp.com/

[12] (2016) Materialise magics. [Online]. Available: http://www.materialise.com/

[13] (2016) Meshlab. [Online]. Available: http://www.meshlab.net/

[14] C. Kirschman and C. Jara-Almonte, "A parallel slicing algorithm for solid freeform fabrication processes," *Solid Freeform Fabrication Proceedings, Austin, TX*, pp. 26–33, 1992.

[15] J. Tölke, "Implementation of a lattice boltzann kernel using the compute unified device architecture developed by nvidia," *Compute Visual Sci*, vol. 13, no. 1, p. 29, 2010.

[16] Z. Shen, K. Wang, and F. Zhu, "Agent-based traffic simulation and traffic signal optimization with gpu," in *14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, Washington, D.C., USA, Oct. 2011, pp. 145–150.

[17] A. Benso and S. Di Carlo and G. Politano *et. al*, "GPU acceleration for statistical gene classification," in *2010 IEEE International Conference on Automation Quality and Teating Robotics*, vol. 2, 2010, pp. 1–6.

[18] R. M. Weiss and J. Shragge, "Solving 3d anisotropic elastic wave equations on parallel gpu devices," *Geophysics*, vol. 78, no. 2, pp. F7–F15, 2013.

[19] P. M. Pandey, N. V. Reddy, and S. G. Dhande, "Slicing procedures in layered manufacturing: a review," *Rapid Prototyping Journal*, vol. 9, no. 5, pp. 274–288, 2003.

[20] C. Dant, "Design and implementation of asymptotically optimal mesh slicing algorithms using parallel precessing," School of Computing at Southern Adventist University, Collegedale, Tennessee, Tech. Rep. DEC-TR-506, Sept. 2015.

[21] M. Vatani, A. Rahimi, and F. Brazandeh, "An enhanced slicing algorithm using nearest distance analysis for layer manufacturing," in *Proceeding of World Academy of Science, Engineering and Technology*, vol. 25, 2009, pp. 721–726.

[22] S. Choi and K. Kwok, "A tolerant slicing algorithm for layered manufacturing," *Rapid Prototyping Journal*, no. 3, pp. 161–179, 2002.

[23] S. McMains and C. Séquin, "A coherent sweep plane slicer for layered manufacturing," in *Proceedings of the fifth ACM symposium on Solid modeling and applications*. Ann Arbor, Michigan, USA: ACM, June 1999, pp. 285–295.

[24] S. Choi and K. Kwok, "A tolerant slicing algorithm for layered manufacturing," *Rapid Prototyping Journal*, vol. 8, no. 3, pp. 161–179, 2002.

[25] R. M. Gregori, N. Volpato, and R. M. *et. al*, "Slicing triangle meshes: An asymptotically optimal algorithm," in *2014 14th International Conference on Computational Science and Its Applications (ICCSA)*, 2014, pp. 252–255.

[26] NVIDIA CUDA. (2016) *NVIDIA CUDA C Programming Guide, Versiion 8.0*. [Online]. Available: http://docs.nvidia.com/

[27] N. Gershenfeld, *Fab: the coming revolution on your desktop–from personal computers to personal fabrication*. Basic Books, 2008.

[28] A. Standard. (2011, July) Standard specification for additive manufacturing file format (amf) version 1.1.

[29] NVIDIA CUDA. (2015) *CUDA Occupancy calculator*. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/

[30] NVIDIA TITAN X. (2015). [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x

[31] Pinshape. (2013) 3d printing community and marketplace. [Online]. Available: http://www.materialise.com/