Towards Scalable and Efficient GPU-Enabled Slicing Acceleration in Continuous 3D Printing

Aosen Wang[1], Chi Zhou[1], Zhanpeng Jin[2] and Wenyao Xu[1]
[1] State University of New York at Buffalo, NY, USA
[2] State University of New York at Binghamton, NY, USA
{aosenwan, chizhou, wenyaoxu}@buffalo.edu, zjin@binghamton.edu

Abstract— Recently, continuous 3D printing, a revolutionary branch of legacy additive manufacturing, has made its two-order time efficiency breakthrough in industrial manufacturing. As its manufacturing technique advances rapidly, the prefabrication to slice the 3D object into image layers becomes potential to impede further improvement of production efficiency. In this paper, we present two scalable and efficient graphic processing unit (GPU) enabled schemes, i.e., *pixelwise parallel slicing* and *fully parallel slicing*, to accelerate the image-projection based slicing algorithm in continuous 3D printing. Specifically, the pixelwise approach utilizes the pixel-level parallelism and exploits the in-shared-memory computing on GPU. The fully parallel method aggressively expands the parallelism on both triangle mesh size and slicing layers. The thread-level priority competing issue, resulting from full parallelism, is addressed by a critical area using atomic operation. Experiments with real 3D object benchmarks show that our *pixelwise parallel slicing* can gain one order of magnitude runtime reduction to CPU, and the *fully parallel slicing* achieves two orders improvement. We also evaluate the scalability of both proposed schemes.

I. Introduction

Continuous 3D printing [1] is an emerging technique in industrial manufacturing. The light polymerized shaping on image projection empowers it dramatically fast speed to print 3D objects. It lowers the production cost and reduces the material waste by using liquid printing materials. The topology correctness guarantee also makes complex 3D topology feasible, which is a challenge in legacy mechanical manufacturing. However, as the rapid advances in manufacturing technique [2], the prefabrication becomes a potential bottleneck for printing efficiency.

Figure 1 illustrates the structure of a continuous 3D printer [1]. The system is divided into two sections: a "wet" part, which is a Carbon3D machine, that performs the chemical, mechanical operations to fabricate a 3D object from a liquid material; and a "dry" part, which consists of a computing unit (e.g., a microcontroller, CPU or FPGA) that sends layer-image based instructions to control the operations on the wet part. The computing process on the dry part is called *prefabrication*. In continuous 3D printing, prefabrication is an image-projection based slicing algorithm extracting the layer information from triangle mesh in a stereolithography format (STL) file [3]. This procedure is a computation-intensive task, where the topology information needs to be retrieved from a sea of distributed triangles. In traditional 3D printing technologies (e.g., stereolithography, inkjet 3D printing), the manufacturing process on the wet part takes a large majority of production time. However, the Carbon3D printer dramatically improves the manufacturing efficiency, and the prefabrication on the dry part is becoming the dominant factor in the production time. For example, a complex multi-scale 3D object (e.g., a bionic vein design with a 7GByte STL file) only takes a few minutes to "print out", while it needs 3-4 hours in the prefabrication process.

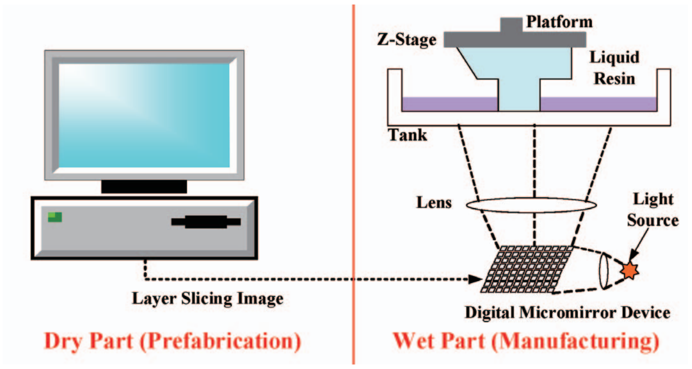

Fig. 1. The entire 3D printing procedure, including a dry part (prefabrication) and a wet part (manufacturing).

In this paper, we present scalable and efficient GPU-based schemes to accelerate the prefabrication of continuous 3D printing. Specifically, we propose two designs, i.e., a pixelwise parallel scheme and a fully parallel scheme, which explore the parallelism enabled by general-purpose GPU (GPGPU) architecture to accelerate the slicing algorithm. The pixelwise parallel slicing utilizes the image pixel level parallelism and unpacks the time-consuming computation in the fast shared memory, effectively reducing the high-delay global memory accessing. The fully parallel slicing further exploits the parallelism on triangle mesh and layer size. An atomic operation based critical area is applied to address the memory access conflict from the multi-thread concurrency. The extensive experiments on public 3D objects indicate that the pixelwise parallel slicing can provide one order of magnitude runtime improvement, and the fully parallel slicing even achieves two orders promotion in the prefabrication of continuous 3D printing. We also evaluate the scalability of proposed approaches.

The remainder of this paper is organized as follows: Section II introduces the background and preliminaries of the slicing algorithm in continuous 3D printing and GPGPU architectures. An analysis of slicing algorithm on CPU is presented in Section III, and two proposed GPU-based slicing acceleration strategies are illustrated in Section IV. The experiments and evaluation are discussed in Section V. Section VI elaborates the related work. Section VII is the conclusion of the paper and our future work.

II. Background

A. Slicing Algorithm for Continuous 3D Printing

The image-projection based slicing method [4, 5] is mainly adopted in continuous 3D printing. Figure 2 shows an illustration example of image-based slicing. The Android man is placed in the 3D space. A cutting plane, perpendicular with z-axis goes through the 3D design and generates its binary image projection in the x-y plane. The black area indicates the pixels inner the Android man body, while the white pixels are outside, which will not be filled with materials. As the cutting plane traverses all the possible z value allowed by manufacturing accuracy, the real Android man is printed.

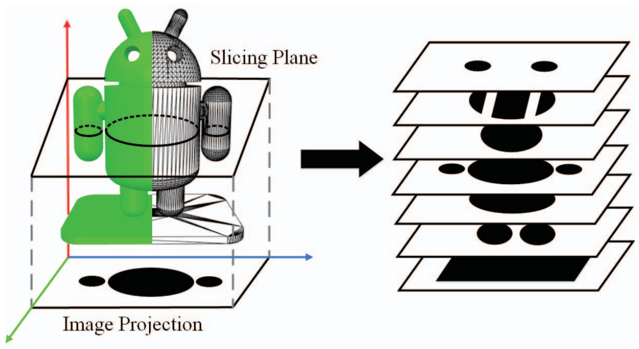

Fig. 2. An illustration example of image-projection based slicing algorithm.

B. GPGPU Architecture in CUDA

The GPGPU architecture [6] is a massively parallel threading processor to run general-purpose applications. Compute unified device architecture (CUDA) [7] is a programming environment to manipulate the GPGPU architecture for intensive computing. Figure 3 depicts the GPGPU architecture.

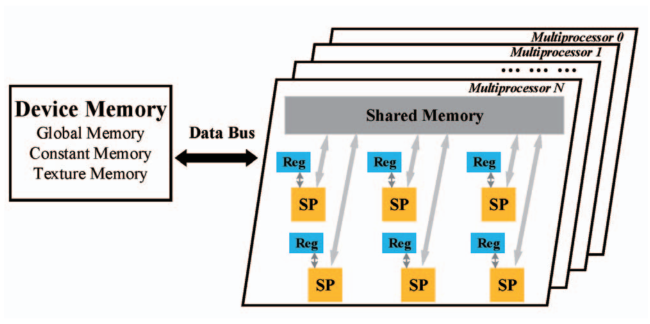

Fig. 3. GPGPU architecture in CUDA.

GPU is a cluster of multiprocessors, each of which consists of many streaming processors (SP), registers and a shared on-chip memory. All the streaming processors in a multiprocessor have their own registers and share a memory with small capacity. All the threads generated in the same multiprocessor can access the shared memory, which is much faster than the on-board device memories. The key factor to improve the performance of GPGPU architecture is to maximize the efficiency of shared memory and registers to reduce runtime by increasing active warp number.

III. Analysis of Slicing Algorithm

In this section, we analyze the structure and procedure of the image-projection based slicing algorithm. As shown in Figure 4, the entire slicing algorithm can be divided into three cascaded functional components: ray-triangle intersection, trunk sorting and layer extraction.

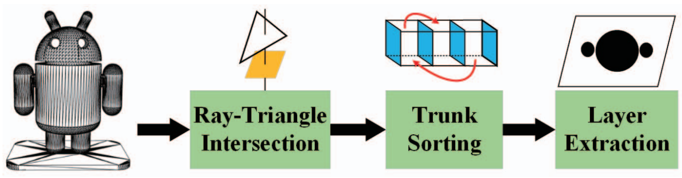

Fig. 4. Block diagram of image based slicing algorithm.

The ray-triangle intersection is to calculate the intersection point between rays on image pixel centers and the triangles from an STL file. For the convenience of extracting layer image, the trunk sorting module sorts the out-of-order intersection points by ascending order in the trunk of each pixel. Then the binary pixel value is identified in the layer extraction module. The detailed description of this sequential slicing algorithm is listed in Algorithm 1.

Algorithm 1 Sequential Slicing Algorithm on CPU

```
Input: STL file $f_{stl}$, projected image resolution $W \times H$ and layer number $L$
Output: A series of binary image $IMG_k$ $(1 \le k \le L)$
1: $(T, TriData) \leftarrow$ ReadSTLFile($f_{stl}$);
2: //Ray-Triangle Intersection
3: $IntersectArray \leftarrow$ newArray3D($W$, $H$,MAXD);
4: for each pixel $pi$ on image do
5:     $(P_x, P_y) \leftarrow$ AxisIdentify($pi$);
6:     for each triangle $tri$ from STL do
7:         $IntersectArray \leftarrow$ Intersect($P_x$,$P_y$,Plane($Tri$));
8:     end for
9: end for
10: //Trunk Sorting
11: for each pixel $pi$ on image do
12:     BubbleSort(trunk on $pi$ from $IntersectArray$);
13: end for
14: //Layer Image Extraction
15: $temp, temp_{old} \leftarrow$ zerosArray($W$, $H$);
16: for each layer $l$ do
17:     $t_{layer} \leftarrow$ LayerHeight($l$);
18:     for each pixel $pi$ on image do
19:         $temp \leftarrow$ updateInfor($t_{layer}$, $pi$, $temp_{old}$);
20:     end for
21:     $IMG_k, temp_{old} \leftarrow temp$;
22: end for
```

The input of the slicing algorithm is a 3D object in an STL file, where T is the triangle mesh size and $TriData$ is the point data. In the ray-triangle intersection step, it first creates a 3D matrix $IntersectArray$ to store the intersections with the size of $W \times H \times$MAXD, where MAXD is a pre-defined trunk depth, a small number practically. *Note that the vector on each pixel is called a trunk.* We use a simple x-y bound test to choose the possible intersecting candidates. If the ray axis is inside the circumscribed rectangle of the specific STL triangle, the test result is true, otherwise, returns false. For the candidates, we continue to check if the ray is falling in the triangle on the x-y plane projection. We can compute the z-axis value of the intersection by analytic geometry.

Now we have all the intersection points for the full resolution of image projection. Since they are out of order in each trunk due to the random triangle distribution in STL file, our next step is to sort the intersection points by ascending order using the bubble sorting [8] on each pixel ray, called trunk sorting.

Finally, we extract all layer images for manufacturing in continuous 3D printing. The binary value of each pixel is dependent on the layer height and the intersection point position in each trunk. For each intersection point whose z value is less than the current layer height, if the normal vector is less than zero along the z direction, we increase the corresponding pixel with one unit, indicating that the ray has entered the inner space of 3D object. Otherwise, we subtract one unit, indicating the ray has gone through the 3D object. Therefore, we can obtain the binary topology image of slicing. In this procedure, we use incremental updating based on the previous layer extraction.

On the basis of the above analysis, there are several challenges to accelerate this slicing algorithm. First, the slicing algorithm is sequential in nature, and three modules are cascaded. It is difficult to implement it in a parallel fashion. Second, due to the high data dependency, it is challenging to design an efficient pipeline between trunk sorting and layer extraction. Third, the computation flow in slicing is complicated, which is not convenient to hardware implementation. In next section, we explore the intra-module parallelism on GPGPU architecture to accelerate the slicing algorithm.

IV. GPU-Enabled Parallel Slicing

We present two GPU-enabled acceleration approaches, pixelwise parallel slicing (PPS) and fully parallel slicing (FPS).

A. Pixelwise Parallel Slicing

In this section, we explore the parallelism inside the slicing algorithm of its GPU implementation. We observe that the operations in all three modules are pixel-independent, i.e., the calculation can be executed independently without complex neighborhood processing. Therefore, we propose a pixelwise parallel slicing, where we assign each pixel ray one specific thread and all the pixel-ray operations are executed by this thread. Our design unpacks all computation tasks in shared memory to reduce interaction with global memory. Figure 5 illustrates the detail of the pixelwise parallel slicing method.

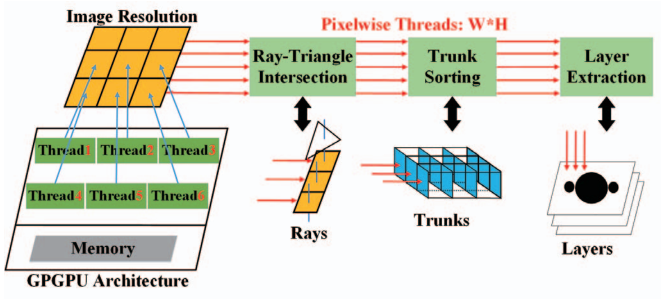

Fig. 5. Block diagram of pixelwise parallel slicing.

The GPGPU architecture can support massive threads whose number is equal to the image resolution $W \times H$. We build a one-to-one mapping from GPU threads to image pixels. Then all the threads start to calculate the ray-triangle intersections. On each pixel, the corresponding ray checks with STL triangles possibly having the intersection. The intersections are stored in the shared memory, where we limit the threads in a GPU block no more than a certain number to ensure the enough space for intersections. After ray-triangle calculation, all threads move to the trunk sorting step. Due to the limited element size in each trunk, each thread directly uses bubble sorting in the shared memory, for the sake of efficient space complexity. Finally, each thread computes binary pixel values for all the layers serially and stores the layer image to the global memory.

B. Fully Parallel Slicing

The PPS method explores the pixel-centric parallelism due to the independent intra-pixel operations. If we take a close look at the slicing algorithm, we note that the PPS still has serial computing components, such as the triangles enumerating in ray-triangle intersection and layer increment in layer image extraction. To utilize the massive thread concurrency of GPGPU architecture and be compatible with a large-size slicing problem, we propose the fully parallel slicing method as Figure 6.

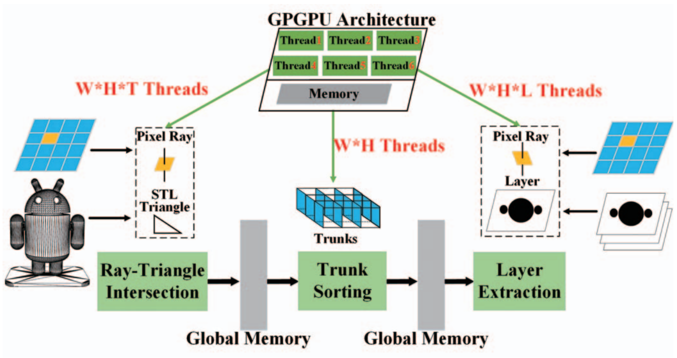

Fig. 6. Block diagram of fully parallel slicing.

Different from PPS, the goal of FPS is to unfold the serial recycles to parallel process. The GPU uses three kernels to calculate the three modules and layer extraction, because of different threads parallelism requirements. In the ray-triangle intersection, we allocate $W \times H \times T$ to cover all the combination between each pixel ray and its corresponding triangle possibly interacted. All the intersections are kept in the global memory. This enables the possibility to solve slicing problem with large-size models. However, this also induces the memory writing conflict when storing intersections to the same trunk from different threads, which we will discuss in section B.1. The trunk sorting is processed as the PPS procedure. In the layer extraction, we also parallelize the computing for every pixel in each layer simultaneously by using $W \times H \times L$ threads, where L is the total layer number. The binary pixel value forms the layer image in the global memory.

B.1 Atomic Operation based Critical Area

To address the issue of multi-thread memory writing conflict, we adopt a customized critical area to solve the conflict, as shown in Algorithm 2. The critical area is using "lock" to guarantee the operation order. Our implementation is particularly designed for parallel programs, avoiding deadlock pitfalls. We

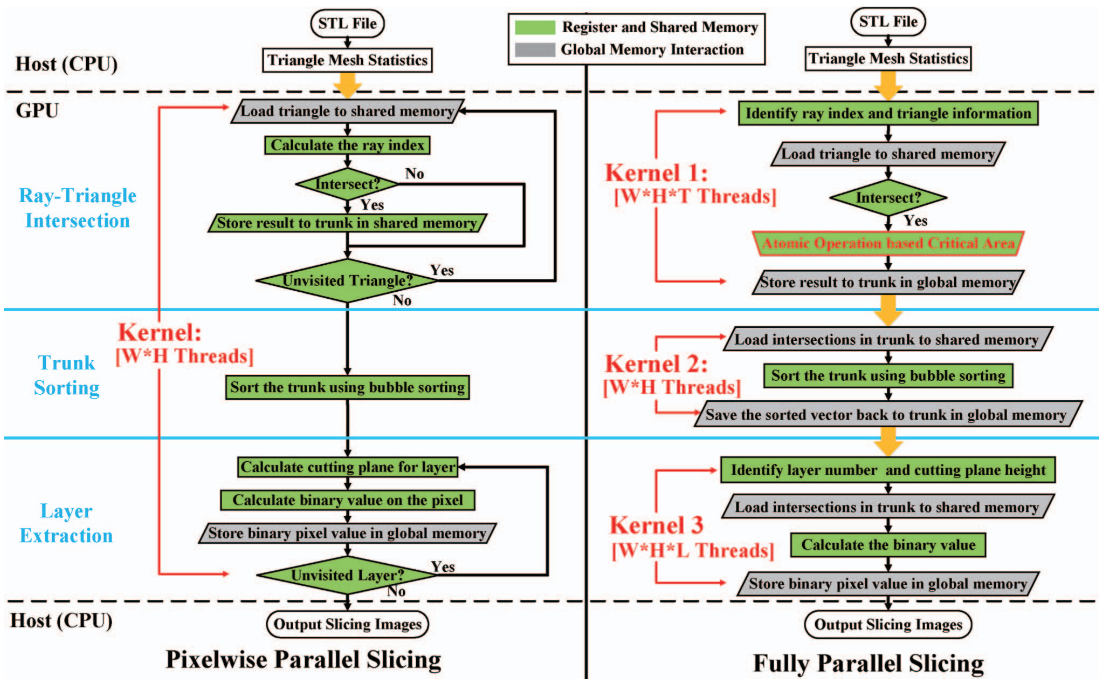
Host (CPU)
GPU
STL File
Triangle Mesh Statistics
Load triangle to shared memory
Calculate the ray index
Intersect?
No
Yes
Store result to trunk in shared memory
Unvisited Triangle?
Ray-Triangle Intersection
Kernel: [W*H Threads]
Trunk Sorting
Sort the trunk using bubble sorting
Layer Extraction
Calculate cutting plane for layer
Calculate binary value on the pixel
Store binary pixel value in global memory
Unvisited Layer?
Output Slicing Images
Pixelwise Parallel Slicing
Register and Shared Memory
Global Memory Interaction
Kernel 1: [W*H*T Threads]
Identify ray index and triangle information
Atomic Operation based Critical Area
Store result to trunk in global memory
Kernel 2: [W*H Threads]
Load intersections in trunk to shared memory
Save the sorted vector back to trunk in global memory
Kernel 3 [W*H*L Threads]
Identify layer number and cutting plane height
Calculate the binary value
Fully Parallel Slicing

Fig. 7. Flow chart comparison between pixelwise parallel slicing (PPS) and fully parallel slicing (FPS).

first initialize locks for each ray, i.e., the input $Lock$ array is of size $W \times H$. In the multi-thread processing, we start with an arbitrary thread with the number id. If the ray intersects with the triangle, it needs to compete for the operation priority. If it fails, the thread will wait for the next iteration to compete the resource. Otherwise, the thread with the priority will lock the trunk on the corresponding pixel, as Line 5 shows. The "atomicCAS(*address*, *compare*, *val*)" is an atomic API from CUDA, which means if the value of *address* is equal to *compare*, the value will be updated with *val*, otherwise, the value keeps the same. Then, we stop the `while` loop in Line 10 and release the lock in Line 11. This method can effectively solve the multiple thread accessing conflict issue.

Algorithm 2 Atomic Operation based Critical Area

```
Input: Lock state array $Lock$ and trunk size array $CntSize$
Output: Intersection points array $IntersectArray$.
 1: for Each thread $id$ do
 2:     if Intersect(Ray(id),Plane($Triangle(id)$)) then
 3:         $run \leftarrow$ true;
 4:         while $run$ do
 5:             if atomicCAS($Lock$[id],0,1)==0 then
 6:                 $cp \leftarrow CntPos$[id];
 7:                 Update($IntersectArray$);
 8:                 $CutPos$[id] $\leftarrow cp + 1$;
 9:                 $run \leftarrow$ false;
10:                 atomicExch($Lock$[id],0);
11:             end if
12:         end while
13:     end if
14: end for
```

C. Comparison between PPS and FPS

We provide a detailed flow chart to compare the two proposed GPU-based slicing implementations, as shown in Figure 7. We can observe the PPS has multiple iterations in the ray-triangle intersection and layer image generation. In contrast, the FPS is a direct processing flow without any iteration. On the other hand, the PPS only creates one kernel to complete all three tasks within the shared memory, while the FPS employs three kernels to complete the slicing procedure, resulting in considerable interaction with the global memory.

Both methods have their own advantages to accelerate the slicing process. The PPS can save the extra addressing computation, because the threads number can be assigned to the pixel position. Considering that all the tasks are completed in the shared memory, PPS avoids the large global memory accessing delay. Moreover, there is no memory accessing conflict among massive threads owing to the characteristics of the image based slicing algorithm. The fully parallel slicing is an aggressively parallel paradigm on the multi-threading platform, which can drastically promote the efficiency of GPU. It empowers recycle-free processing for the image projection based slicing algorithm. The thread-level accessing conflict can be solved by the critical area with some resource and speed compromise. In the next section, we will evaluate the performance of PPS and FPS through experiments.

V. Experiments

In this section, we carry out a set of experiments to examine the efficiency and scalability of two GPU-based slicing approaches. We first describe the experimental setup. Then we compare the run time of benchmarks under a typical parameter configuration. Scalability with different projected image resolutions and layer numbers are also evaluated.

A. Experiment Setup

We choose four representative 3D objects as our benchmarks to examine the performance of CPU and GPU implementations, i.e., Club, Android, Ring and Bunny, whose 3D views are shown in Figure 8. The four objects have an increasing triangle mesh size by the order from Club to Bunny. The Club has 3290 triangles, Android has 10926, Ring has 33730 and Bunny is with 69664 triangles. To obtain the detailed performance, we use cycle-accurate CPU and GPU simulators to testify the performance of CPU implementation and our two GPU designs in the experiment. Specifically, we deploy Sniper [9] to run the sequential image-projection based slicing algorithm. We configure the CPU in Sniper as the typical Intel Xeon X5550, which is a four-core general-purpose processor with 2.66 GHz frequency. GPGPU-Sim [10] simulator is adopted to test the performance of our PPS and FPS methods. It is configured as a typical Nvidia GeForce GTX480, which has 480 streaming processors with 700 MHz processing frequency. The both CPU and GPU simulators have been calibrated with real processors. For pixelwise parallel scheme on GPU, we set its block size equal to the height of projected image and the grid size as image width. While in fully parallel method, the block size is (16, 16) and grid size is also a two-dimension vector with equal elements, whose value is the square root of total threads over 256.

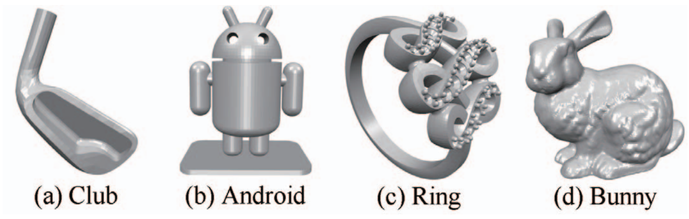

Fig. 8. 3D views of the four representative benchmarks.

B. Runtime Comparison

First, we evaluate the typical runtime comparison among the four benchmarks. We choose the projected image resolution as 256×128, and the total cutting layer number L is 100. We collect the total cycles for three components separately in Sniper. We directly run GPU applications under the CUDA environment and obtain the cycle statistics by GPGPU-Sim. The comparison of total cycle results are plotted in Figure 9.

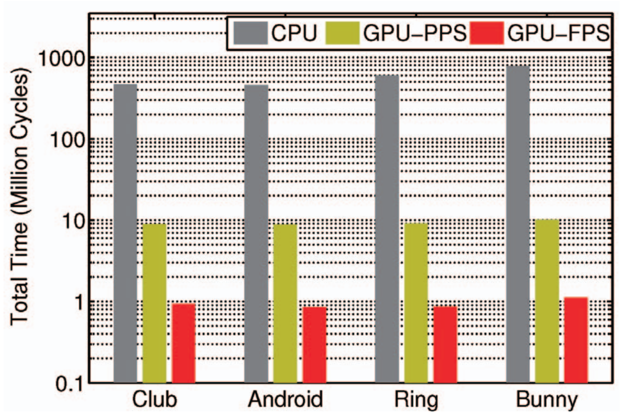

Fig. 9. Total time comparison of four benchmarks on CPU and GPU.

For clarity of the presentation, we use the logarithmic scale to represent the total time. With the processing frequency of the typical computing platforms, we can obtain the total runtime estimation on a real processor. GPU-FPS achieves the best performance in the three cases, promoting approximately two orders improvement than CPU runtime. The pixelwise parallel slicing gains about one order of magnitude speedup than CPU. Its pixel-level parallelism and in-shared-memory processing reinforces it the huge advantage of acceleration compared with CPU implementation. However, due to the partially serial processing and not so strong processor architecture as CPU, it only obtains $13.77\times$, $13.73\times$, $17.18\times$ and $20.44\times$ speedups for the four benchmarks, respectively. When it comes to the fully parallel slicing, its recycle unfolding effectively makes GPGPU architecture running in its peak state for the large triangle number and layer number. *The FPS excavates the parallelism of massive-threading architecture better, though it has a more frequent global memory accessing pattern than the PPS case.* Eventually, it gains $132.64\times$, $141.51\times$, $180.92\times$ and $184.15\times$ for the four benchmarks, about one order of magnitude superior to the PPS case. We can also find the performance of both GPU accelerations increase as the input triangle mesh size grows.

For the CPU case, the total cycles are not showing a rapid increasing as the input triangle number increases. This is because our possible intersection check can filter out a lot of simple cases, whose axis in x-y plane cannot fall in the circumscribed rectangle of the specific STL triangle. For the Android case, its runtime is a little smaller than the Club. This is because the regular space topology of Android reduces more intersection candidates.

C. Scalability on Projected Image Resolution

Second, we examine the scalability of our GPU approaches on different projected image resolutions. The image resolution is the only variable affecting the performance of all three modules in the slicing algorithm. We choose three image resolutions, 128×64, 256×128, 512×256, and take all the four benchmarks in this experiment. Sequential slicing algorithm is still running in sniper and our GPU slicing methods on GPGPU-Sim. We illustrate the speedups of pixelwise parallel slicing (GPU-PP) and fully parallel slicing (GPU-FP) to CPU case in Figure 10.

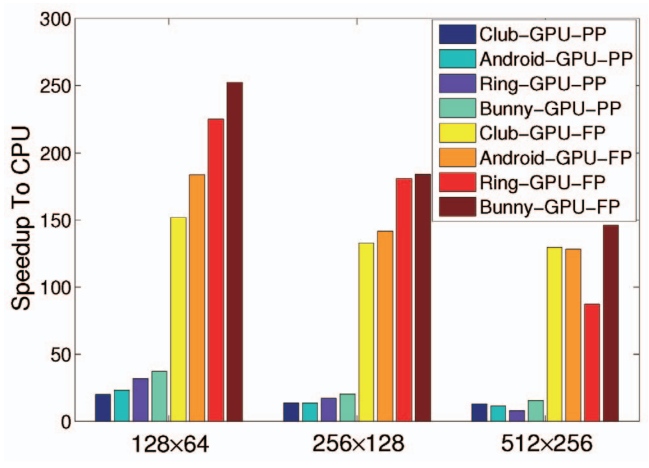

Fig. 10. Speedups to CPU of benchmarks under different image resolutions.

All the cases are consistent with the typical case, where the fully parallel slicing method gains about two orders improvement and the pixelwise parallel slicing has more than one order of magnitude performance. If we take a close look at Figure 10, we find that the speedup has a decreasing trend as the resolution increases. The objects with larger size triangle mesh have bigger speedup degradation. Moreover, the fully parallel slicing decreases more than the pixelwise parallel slicing. In fully parallel strategy, the GPU active warps tend to full-load working and the dramatically increasing tasks must wait for the idle warp. The pixelwise method has only image-resolution threads, which can be handled well by GPU architecture without active warp competing. This firmly demonstrated the scalability of our two GPU implementations on image resolution.

D. Scalability on Layer Number

At last, we evaluate the scalability of our GPU acceleration on different layer numbers. The layer number is a significant factor to influence the efficiency of layer image extraction module. We choose one benchmark Bunny and set the cutting layers as three levels: 10, 100 and 1000. The pixelwise parallel slicing is difficult to provide accurate cycle information of submodules due to one kernel integration. Therefore, we compare the total cycles of sequential slicing on CPU and fully parallel slicing on GPU. The run time statistics are listed in TABLE I.

TABLE I
The comparison between CPU and GPU on layer number.

Time (Million Cycles)		Layer Number		
		10	100	1000
CPU	Ray-Triangle	517.38	517.34	517.51
	Trunk Sorting	12.06	11.99	12.08
	Layer Extract	**33.11**	**255.32**	**2459.17**
	Total	562.55	784.65	2988.76
GPU	Ray-Triangle	0.300	0.301	0.300
	Trunk Sorting	0.020	0.020	0.020
	Layer Extract	**0.084**	**0.800**	**7.965**
	Total	0.404	1.121	8.285

Based on the clock frequency of CPU and GPU, we can calculate the runtime speedups as 366.47×, 184.15× and 94.93× for the three layer number levels, respectively. Our fully parallel slicing still achieves about 100× speedup under all cases. When layer number is small, the ray-triangle intersection takes up the majority runtime in both CPU and GPU cases. As layer number increases, the layer extraction consumes more time. CPU needs more time to sequentially calculate the image extraction with the much larger resolution by traversing every pixel. The numerous global memory accessing in the fully parallel GPU implementation degrades the speedup improvement. Another observation is that the trunk sorting only has a subtle weight in the total runtime, though we use basic bubble sorting. Therefore, the GPU accelerations can still provide high performance on the slicing task with increasing layer numbers.

VI. Related Work

An early slicing algorithm working on an STL file was an adaptive approach [11]. Then several work attempted to improve the time efficiency by avoiding unnecessary operation overhead. Chakraborty *et al.* developed a semi-analytical method to enhance the efficiency of surface plane intersection calculation [12]. Sun *et al.* designed an adaptive slicing scheme [13] to adapt the layer thickness based on curvature information to promote the slicing efficiency. However, all these attempts are working on contour-oriented methods, while the continuous 3D printing processes projected image. Since continuous printing concept is the latest breakthrough [1], there is still no in-depth work to address its computation issue. Therefore, there is an urgent need to have a slicing acceleration scheme that can provide orders of magnitudes efficiency improvement to keep pace with continuous 3D printing.

VII. Conclusion and Future Work

In this work, we investigated the slicing algorithm acceleration on GPGPU architecture for continuous 3D printing. We first analyzed the sequential slicing algorithm on CPU. To explore the pixel-level parallelism and better usage of the precious shared memory and registers of GPU, we proposed a pixelwise parallel slicing implementation. Optimizing the parallelism further, we proposed fully parallel slicing scheme. We also used an atomic operation based critical area to solve the memory accessing conflict among multiple threads concurrency. The experiments indicated our pixelwise parallel slicing can achieve one order runtime improvement and the fully parallel slicing even gains two orders performance promotion. The scalability of the both GPU implementations is also verified.

In the future work, we consider designing new methods to further accelerate the prefabrication based on the new hardware platform. On the other hand, we also plan to explore the pipeline property between the prefabrication and manufacturing for better time efficiency in continuous 3D printing.

Acknowledgments

This work is in part supported by National Science Foundation grants CNS-1423061/1422417, ECCS-1462498/146247, CNS-1564104/1564046 and CNS-1547167.

References

[1] John R Tumbleston, David Shirvanyants, Nikita Ermoshkin, Rima Janusziewicz, Ashley R Johnson, David Kelly, Kai Chen, Robert Pinschmidt, Jason P Rolland, Alexander Ermoshkin, et al. Continuous liquid interface production of 3d objects. *Science*, 347(6228):1349–1352, 2015.

[2] Fan Yang, Feng Lin, Chen Song, Chi Zhou, Zhanpeng Jin, and Wenyao Xu. Pbench: a benchmark suite for characterizing 3d printing prefabrication. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pages 1–10. IEEE, 2016.

[3] Eric Béchet, J-C Cuilliere, and François Trochu. Generation of a finite element mesh from stereolithography (stl) files. *Computer-Aided Design*, 34(1):1–17, 2002.

[4] Chi Zhou, Yong Chen, Zhigang Yang, and Behrokh Khoshnevis. Digital material fabrication using mask-image-projection-based stereolithography. *Rapid Prototyping Journal*, 19(3):153–165, 2013.

[5] Yong Chen and Charlie CL Wang. Layer depth-normal images for complex geometries: Part one?accurate modeling and adaptive sampling. In *ASME 2008 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 717–728. American Society of Mechanical Engineers, 2008.

[6] Enhua Wu and Youquan Liu. Emerging technology about gpgpu. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 618–622. IEEE, 2008.

[7] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[8] S Hutzler, D Weaire, and S Shah. Bubble sorting in a foam under forced drainage. *Philosophical magazine letters*, 80(1):41–48, 2000.

[9] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2011.

[10] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

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

[12] Debapriya Chakraborty and Asimava Roy Choudhury. A semi-analytic approach for direct slicing of free form surfaces for layered manufacturing. *Rapid Prototyping Journal*, 13(4):256–264, 2007.

[13] SH Sun, HW Chiang, and MI Lee. Adaptive direct slicing of a commercial cad model for use in rapid prototyping. *The International Journal of Advanced Manufacturing Technology*, 34(7-8):689–701, 2007.