Revisiting Slicing for High Resolution Continuous 3D Printing

***Abstract*—While conventional 3D printing has largely focused on low resolution applications (e.g., cm and mm scale resolutions), there is now significant interest in fabrication at micro and nanoscale resolutions for a wide variety of emerging applications. We argue that slicing overhead may become unacceptable at these high resolutions. The high slicing overhead will be due to the large size of the slicing output generated at these resolutions (often too large to fit into memory!). It will also be due to the poor work efficiency of existing slicing algorithms. To address these issues, we propose a new output format for slicing. The new output format reduces the size of the sliced output significantly (by 233x-1519x). We also present a new slicing algorithm that significantly reduces wasted work (by 29x-948x). Together, the optimizations allow slicing for models that previously could not be sliced on a state-of-the-art GPU. For other models, we see significant performance speedups (by 20x-378x).**

***Index Terms*—Additive manufacturing, 3D printing, slicing**

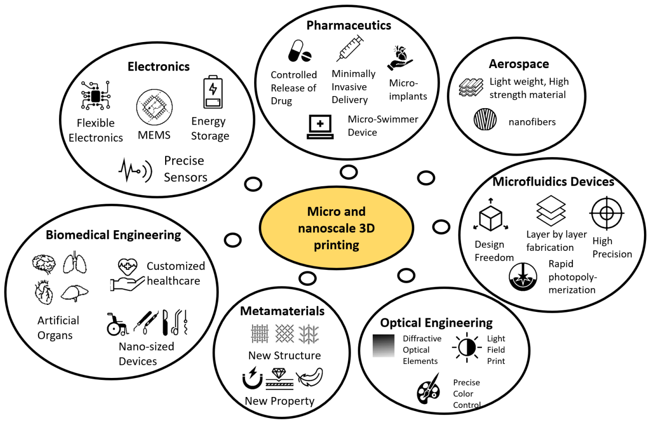

Fig. 1. Applications of micro and nanoscale 3D printing.

I. Introduction

Additive Manufacturing (AM), also known as three-dimensional (3D) printing, has drawn increased attention and usage in a wide range of fields, including aerospace [1], biomedical engineering [2], medicine [3], automobile [4] and architecture [5]. Benefits of 3D printing include high design freedom and customizability [6], fast prototyping [7], and low cost [8]. Unsurprisingly, its popularity and applications have been exploding.

While conventional 3D printing has largely focused on low resolution applications (e.g., cm and mm scale resolutions), there is now significant interest in fabrication of complex three-dimensional structures at micro and nanoscale resolutions in areas such as novel materials, electronics, biomedical engineering, micro fuel cells, and optical engineering [9]. In Figure 1, we summarize some applications of high resolution 3D printing. Examples include materials with ultrahigh mechanical efficiency [10], flexible electronics [11], micro batteries for energy storage to improve energy density [12], biopolymers to build synthetic scaffolding for cell growth [13], organs-on-chips [14], [15], near-perfect Diffractive Optical Elements (DOEs) for precise wavefront shaping [16], high-resolution light field prints (LFP) to overcome pixelation limitations [17], pharmaceuticals [18], controlled release [19], minimally invasive [20] and high-precision [21] drug delivery, biomimetic models for drug discovery and development [22], and resilient and lightweight components for the aerospace industry [23].

In this paper, we argue that high resolution 3D printing may have vastly different computational tradeoffs than conventional 3D printing. The 3D printing process can be divided into two parts. The first part is a computing process called pre-fabrication or process planning [24]. A 3D printable model either designed through Computer-Automated Design (CAD) or generated through 3D scanning is presented as triangle meshes in Standard Triangle Language (STL)[1] format and then converted into sliced layers with G-code[2] to instruct a 3D printer. The prefabrication steps include image-based slicing, path planning, support generation, orientation, repairing and packaging [25]. The second part is manufacturing stage, in which the printer fabricates the physical objects. Prefabrication time is already significant today for increasingly large and complex 3D models, sometimes dominating the overall 3D printing process [25]. In particular, slicing can be very computationally expensive [25]. Slicing converts 3D models, which consist of triangles connecting at the vertices, into 2D layers. Slicing time, therefore, is highly dependent on model size, which is reflected by the number of triangles. It also depends on complexity of the model, which is reflected by the dimensions of triangles and the resolution required to fully describe model information. We argue that as we move towards micro and nanoscale resolutions, the effective model size and model complexity will become much larger (more triangles, smaller triangles), thus vastly increasing the computational demand of slicing. Higher resolution will also cause more pixels and layers, increasing the possibility of repeatedly visiting one triangle during slicing, thereby exacerbating the slicing computational overhead further. Our work focuses on understanding and reducing slicing overhead for high resolution 3D printing.

In particular, we focus on slicing for high resolution *continuous* 3D printing [26] - a variation of legacy stereolithography (SLA)-based additive manufacturing that supports much higher printing speed and higher resolution [27]. Unlike conventional 3D printing technologies where models are printed by printers “drawing lines” on every layer, models are “grown” out of printing materials in continuous 3D printing. The key idea behind a canonical continuous 3D printer is: when stereolithography is conducted above an oxygen-permeable build window (Figure 2), continuous liquid interface production (CLIP) is enabled by creating an oxygen-containing “dead zone”, a thin uncured liquid layer between the window and the cured part surface. Because of this “dead zone”, models “grow” from a tank of liquid material (usually resin), unlike traditional stereolithography printers, where UV exposure, resin renewal, and part movement must be conducted in separate and discrete steps. Because of the continuous

[1]STL is a file format created by 3D Systems. An STL file describes a raw, unstructured surface by an unordered list of triangle facets and the unit normal. The vertices of triangles are ordered by the right-hand rule.

[2]G-code is a widely used computer numerical control (CNC) programming language. G-code instructions are provided to a machine controller (industrial computer) that tells the motors where to move, how fast to move, and what path to follow.

growing process, print speed of CLIP increases by at least an order of magnitude while maintaining excellent part accuracy [28]. Initially the build plate is at the bottom of the tank and the binary image of the bottom layer is projected onto the bottom of the liquid (usually using UV light) through the oxygen-permeable. The liquid at the bottom is solidified by the image projection as the build plate moves up at constant speed. Every time the build plate moves up by the height of a layer, the printed layer is attached to either the build plate or the previous layer and the projected image is replaced by the image of the next layer.

Unlike conventional 3D printing techniques such as fused deposition modeling (FDM), stereolithography (SLA) and selective laser melting (SLM) which often have insufficient resolutions (over 100 μm) to meet the demands for micro and nanoscale 3D structures [29], continuous 3D printing can be used even for sub-micron printing. In fact, continuous 3D printing is widely used today for micro and nanoscale 3D printing [30] [31] [32]. Commercial continuous 3D printers [33], [34] also tend to be over 30 times faster than conventional printers. Besides, the print speed of a continuous printer does not typically depend on resolution [28], while the print speed of a conventional FDM-like printer is lower at higher resolution. Due to these factors, continuous printing is becoming a de facto standard for printing at high resolution.

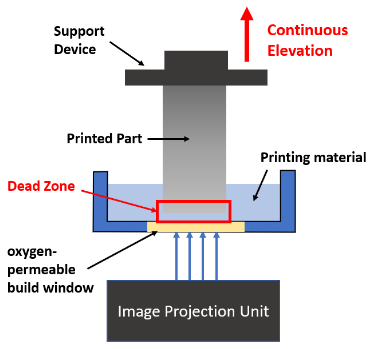

Fig. 2. An example of continuous printing: the liquid printing material is kept in a tank with an oxygen-permeable build window at the bottom; an image projection unit solidifies the printing material at the bottom of the tank; the printed part is attached to a support device (build plate) which is continuously elevating.

In this paper, we show that slicing can have unacceptable overhead for continuous 3D printing at high resolutions (Section III). We discover that there are two primary overheads. First, the output of slicing is large (often too large to fit into memory!) which could not only make the overhead of data movement between the device running the slicing algorithm (e.g., a GPU) and the host device (e.g., a CPU) large, but also makes slicing at very high resolutions hard due to insufficient memory. Second, prior slicing algorithms have poor work efficiency, i.e., most of the work is discarded. To address the first problem, we propose a new output format for slicing (Section IV). The new output format reduces the size of the sliced output significantly (by 233x-1519x). To address the second problem, we present a new slicing algorithm (Section V) that significantly reduces wasted work (by 29x-948x). Together, the optimizations allow slicing for models that previously could not be sliced on a state-of-the-art GPU. For other models, we see significant performance speedups (by 20x-378x).

This paper makes the following contributions: (1) Experimental analysis showing that slicing overhead may be unacceptable at high resolution due to large output size and poor work efficiency. (2) A new output format for image-projection-based slicing that reduces memory requirement and enables slicing at very high resolution. (3) An efficient slicing algorithm for slicing large models at high resolution.

II. Background and Related Work

Prior work on slicing algorithms has focused on two classes. The first class of slicing algorithms uses a contour-generation based slicing method. This method of slicing characterises the 2D layer information by using perimeter boundaries which are closed loops of connected segments [35]. Contours are extracted by connecting line segments which are the intersections of triangles and Z planes (Figure 3). After slicing, path planning is needed to fulfill the contour to tell printer where to go to generate this layer. Contour-generation based slicing is mainly adopted in technologies which require movement of nozzles like Fused Deposition Modeling (FDM).

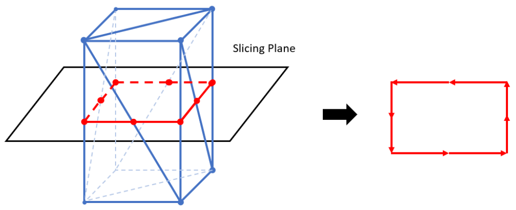

Fig. 3. Contour-generation based slicing: calculate line segments first, which is intersections between the slicing plane and all triangles intersecting with this slicing plane (red lines in the left part). Then connect these line segments end to end to form a polygon, which is the contour to characterise the model on this plane, as shown in the right part.

The second class of slicing algorithms, primarily used in context of continuous printing, is based on image projection [36]. Figure 4 illustrates the idea of image-projection based slicing. The model to be sliced is placed in the build volume. A slicing plane is a plane perpendicular to the Z axis. For each Z value supported by the printer, the slicer needs to generate a binary image of the corresponding slicing plane. In the binary images shown in Figure 4, the black pixels are inside the model and the white pixels are outside. When printing, the printer traverses these images in order and fills the inside of the model with material. Therefore, the goal of slicing for continuous printing is to extract the state of every pixel (inside or outside) in every layer supported by the printer.

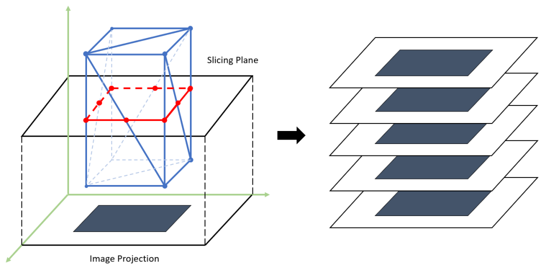

Fig. 4. Image-projection based slicing: the slicing software calculates the state of every pixel on slicing plane (whether it is inside or outside the model) to generate a projected image of the model for every layer. Here, the black part in a certain layer is the set of pixels that are inside the model, and it need to be fulfilled in fabrication stage.

Recognizing the high computational overhead of slicing, both classes of slicing algorithms have seen GPU and parallel implementations [37] [38]. Pixelwise Parallel Slicing (PPS), proposed by Wang et al. [38] focuses on image-projection slicing. It is the baseline for our work. It consists of three steps: *ray-triangle intersection*, *trunk sorting* and *layer extraction*. The ray-triangle intersection module calculates the intersection point between a ray, which is the set of pixel center points with same x-y coordinate but different Z values, and triangle meshes in STL input. The trunk sorting module sorts the out-of-order intersection points by ascending order in a trunk. The layer extraction module calculates the binary value of each pixel and generates the projected image for printing. Algorithm 1 and 2 show this process. Threads in a GPU are allocated to each pixel-ray to perform all the pixel-ray operations as shown in Figure 5.

Algorithm 1: PPS: Ray-Triangle Intersection

```
Input: allTriangles
Output: interesectionTrunk
/* Get coordinates from thread and block
   index                                          */
$pixelRayCoords \leftarrow (threadX, threadY)$;
/* Iterate over all triangles in the model
   */
foreach tri in allTriangles do
    $intersection \leftarrow$
     computeIntersection($pixelRayCoords, tri$);
    if intersection is valid then
        /* Only store the z value of
           intersection, because x and y can
           be derived from thread and block
           index                                  */
        add $intersection$ to $intersectionTrunk$;
    end
end
```

Algorithm 2: PPS: Trunk Sorting & Layer Extraction

```
Input: intersectionTrunk
Output: slicedTrunk
/* Sort the intersections in ascending
   order by z values                              */
$sortedTrunk \leftarrow$ sort($intersectionTrunk$);
append $numLayers$ to $sortedTrunk$;
$idx \leftarrow 0$;
for $z = 0$ to $numLayers$ do
    /* Find the number of intersections
       below the current layer                    */
    while $sortedTrunk[idx] < z$ do
        $idx \leftarrow idx + 1$;
    end
    /* A pixel is inside the model if it
       contains an intersection, or if
       there are odd number of
       intersections below it                     */
    $hasIntersection \leftarrow (z == sortedTrunk[idx])$;
    $isInside \leftarrow (idx\%2 == 1)$;
    $slicedTrunk[z] \leftarrow (hasIntersection || isInside)$;
end
```

Calculation of ray-triangle intersection in PPS may seem similar to ray tracing at high level. However, we only want to find the first intersection between an arbitrarily ray and a 3D object in case of ray tracing . On the other hand, we want to find intersections for all ray-triangle pairs in case of slicing, which means that a single ray could intersect with the surface of an object multiple times. Also, ray-tracing programs are designed to handle rays in arbitrary directions. In case of slicing, all rays are perpendicular to the XY plane. In general, the ray-triangle intersections in slicing are much easier to compute than arbitrary ray-triangle intersections in ray tracing.

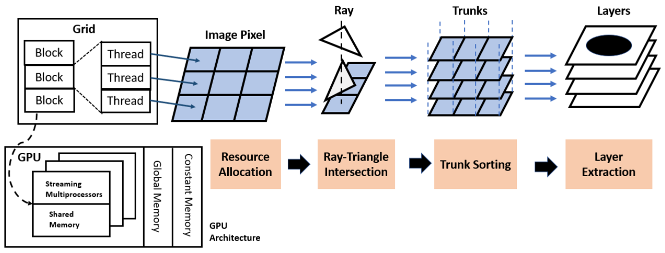

Fig. 5. Block diagram of pixelwise parallel slicing: PPS consists of a setup step (resource allocation) and three computation steps (ray-triangle intersection, trunk sorting, and layer extraction.) In ray-triangle intersection the valid intersections between all pairs of pixel-rays and triangles are found. Intersections with the same XY values are stored in the same trunk. In trunk sorting these intersections are sorted by their Z values (height). In layer extraction, a projected image is generated for every layer based on these sorted trunks.

III. Motivation

When slicing small models at low resolution, PPS has been shown to be an effective algorithm because it successfully exploits the thread and data-level parallelism in slicing. However, when slicing large models at high resolution, it is unclear if PPS can handle the increased computation and memory requirement. Consider the memory requirement for PPS. PPS generates output in bitmap format. (as any other conventional continuous slicing algorithm). However, bitmaps will be large at high resolution since they store the information of every pixel, and the number of pixels scales cubically with the resolution[3].

To quantify the effectiveness of PPS at high resolution, we measured the time and memory needed to slice the models shown in Table I. *tweel* and *engine* are chosen to be representative of models in manufacturing industry that can be printed at high resolution (using micro and nanosclae metamaterials, for example). *kidney*, *skull*, *skeleton*, and *brain* represent bio-models that may need to be printed at high resolution (for replacement and drug discovery, for example). *Android* and *bunny* were chosen for better comparison against PPS since they were used in the original PPS work [38]. The build volume (printing space in a 3D printer) is set to $128 \times 128 \times 128\text{mm}^3$ and the layer height is assumed to be equal to the XY resolution. Slicing is performed on an NVIDIA TITAN Xp graphics card with specifications shown in Table II. The host CPU we used in these experiments is Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz.

Figure 6 shows the size of the bitmap-based slicing output at different resolutions for the model *skull*. Note the extremely high memory requirement of PPS at high resolution. In fact, since the

[3]A single bitmap stores the state of every pixel on a layer (XY plane) explicitly, therefore it scales linearly with the number of pixels on a layer, or quadratically with the XY resolution. The number of bitmaps needed is the same as the number of Z values supported by the printer, which is inversely proportional to the layer height. The total size scales cubically if we assume the XY resolution is the same as layer height.

Model	File Size	Triangle count	Dimension(mm)
Android [39]	251.1KB	5020	$76 \times 51 \times 93$
bunny [39]	3.5MB	69664	$86 \times 66 \times 85$
tweel [39]	28.3MB	565108	$101 \times 22 \times 101$
kidney [40]	81.1MB	1660268	$86 \times 85 \times 43$
skeleton [40]	101.8MB	2084600	$87 \times 79 \times 125$
engine [39]	116.8MB	2336382	$112 \times 99 \times 123$
skull [40]	170.2MB	3486070	$71 \times 103 \times 116$
brain [40]	317.6MB	6505750	$85 \times 80 \times 117$

TABLE I
BENCHMARK MODELS USED FOR THIS STUDY.

GPU	TITAN Xp
Global memory	12196MB
Shared memory	48KB
Constant memory	64KB
Block registers	65536
Warp size	32
Threads per block	1024
Max block dimensions	$[1024, 1024, 64]$
Max grid dimensions	$[2147483647, 65535, 65535]$

TABLE II
GPU SPECIFICATION

original PPS algorithm keeps the entire output in the memory, it could not slice the model at 31.25µm resolution or beyond. The memory requirement at 31.25µm is about 70GB, which far exceeds the 12GB memory capacity of the GPU we used for our experiments. To enable slicing at higher resolution, we broke the model into different blocks[4] and ran PPS on each block individually. This allowed us to reduce the memory requirement and slice the model at higher resolution. However, this method does not work at 0.98µm resolution and beyond, because even the size of a single block was too large to fit in GPU memory at these resolutions. While one could increase the scalability by allowing blocks to be smaller than a layer, the fundamental problem remains.

Some prior work has considered bitmap compression for large data sets. For example, Mohamed and Fahmy [41] introduce an algorithm for compressing binary images where a binary image is first partitioned into non-overlapping rectangular regions, where each rectangular region contains pixels of the same value. Then, the size, position, and value of all the rectangular regions are stored. Zahir and Naqvi [42] improve this scheme by allowing overlapping rectangles. Unfortunately, these works require uncompressed data as input. Since our output is too large to be generated and stored in memory in the first place for several high resolution values, these compression algorithms cannot be applied[5]. While the problem may be alleviated if the memory size is much larger, we do not expect commercial GPUs to have large enough memory in the near future. Similarly, out-of-core slicing algorithms may alleviate the problem (the problem still remains when the block size becomes larger than memory size), such solutions introduce hardware and software complexity, especially in our GPU-based slicing context.

[4]Each block is a set of layers. For example, we can slice a 128x128x128 model by slicing four 128x128x32 blocks.

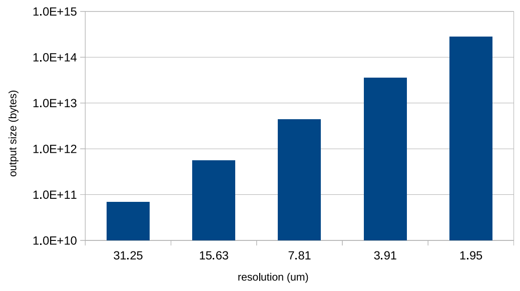

Fig. 6. Output size of model *skull* when using bitmaps at different resolution.

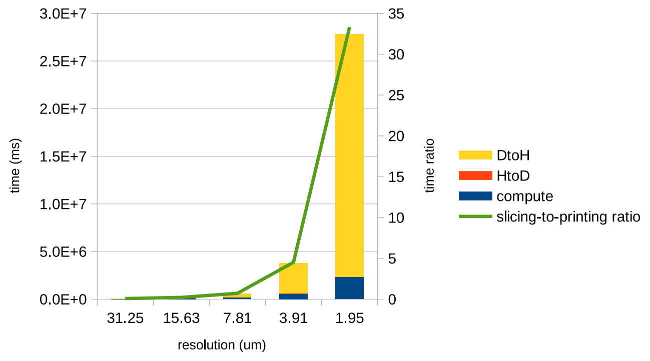

Fig. 7. PPS slicing time of model *skull* as a combination of compute, host-to-device data transfer (HtoD), and device-to-host data transfer (DtoH). The slicing-to-printing ratio is calculated as slicing time divided by (estimated) printing time at different resolutions.

Other than high memory requirements, slicing time at high resolution is also a concern. Figure 7 shows the slicing time for the the same model (*skull*) at different resolutions. Slicing a model using the PPS algorithm involves three steps: (1) copy the model from CPU to GPU memory, (2) run the PPS algorithm (including ray-triangle intersection, trunk sorting, and layer extraction) on GPU, (3) copy the output bitmaps back to CPU memory. The time taken to move a model from CPU to GPU memory depends only on the model size, which is no more than a few hundred megabytes even for the large models we have. So, this component of the time overhead is small at at all resolutions. At low resolution ($\geq$ 15.63µm), the output size is not large. Therefore, the data movement overhead in step 3 is relatively small at these resolutions. On the other hand, the ray-triangle-intersection calculation in step 2 is relatively high overhead at low resolution; therefore it dominates the overall running time at resolutions $\geq$ 15.63µm (even as the overall running time is manageable at these resolutions).

Overall overhead explodes at high resolution ($\leq$ 7.81µm). At 1.95µm, the slicing overhead is over 8 hours! As discussed above, the output size is inversely proportional to the cube of resolution. At high resolution ($\leq$ 7.81µm), the data movement overhead becomes very large and leads to the high overall overhead. As a point of interest, although we were unable to get PPS results for 0.98µm and smaller resolution due to memory limitation, we expect that compared to the 1.95µm case, PPS would take about 8 times longer at 0.98µm resolution if given enough memory, which is 2.23E+5 seconds or 2.6 days for the model we are using.

Figure 7 also shows the slicing time compared to the estimated printing time on a continuous printer with 500mm/hr printing speed. Although slicing time is insignificant at low resolution, it is comparable to printing time at 7.81µm resolution and up to 35 times more

[5]For some high resolutions, where it may be possible to apply GPU-based compression, proposed new output format provides strictly greater performance benefits - Section IV

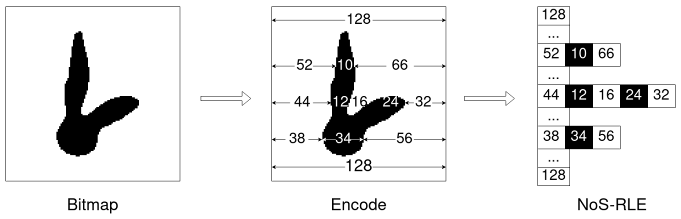

Fig. 8. Convert a bitmap to NoS-RLE format.

than printing time at 1.95μm resolution. This ratio would be even larger for faster printers (for example, HP MultiJet Fusion printer [43] claims to print at the speed of 3000mm per hour on Z axis) and at higher resolutions.

We repeated the above experiment with other 3D models. Step 1 execution time depends on the input model size, but the value is always small compared to the execution time of step 2 and 3. Although the computation time in step 2 is different for different models at low resolution, the difference between models diminishes at high resolution when layer-extraction dominates the computation time[6]. Step 3 only depends on the output size and therefore takes the same amount of time for all models at a given resolution. Overall, the total slicing overhead at high resolution was large and mostly similar across models.

IV. NoS-RLE: A New Format for Slicing Output

We argue that the output of slicing does not have to be bitmaps. A pixel in the build volume can have two states: it can either be inside the model, or outside. A valid slicing output should have the following two properties: (1) the state of every pixel in the build volume should be accurately stored; (2) the state of every pixel should be easily retrievable. These two goals are achieved in traditional slicing by explicitly storing the state of every pixel in bitmaps.

In this section, we present a new output format for image-projection-based slicing: *No Symbol Run-Length Encoding (NoS-RLE).* We believe NoS-RLE is a valid slicing output format for the following reasons: (1) It can be easily converted into bitmaps, so it is compatible with all the continuous 3D printing workflows which expect bitmaps. (2) The state of any pixel can be retrieved with low overhead.

The NoS-RLE format is based on run-length encoding. In run-length encoding, a sequence of symbols is encoded as a sequence of (symbol, run-length) pairs. In the case of slicing output, there are only two states for a pixel (inside or outside the model). Therefore, by assuming some initial state[7], we can encode the sequence using only the run-length values. For example, suppose we have the following two sequences:
(1) 0011 1111 1100 0110
(2) 1110 0110 0001 1100
In basic run-length encoding, these sequences would be encoded as
(1) 02 18 03 12 01
(2) 13 02 12 04 13 02

[6]Most of the time in layer extraction is spent on calculating the correct values for pixels and writing them to memory. The number of pixels in the build volume does not depend on the model (it depends on the printer size instead). While reading the intersections (needed for calculating pixel values) does depend on the model (different models will likely have different number of intersections), it is a lower overhead compared to the rest.

[7]In this paper we consider pixels on the surface to be inside the model. But even if we treat them as outside, the analyses and conclusions would be the same.

In NoS-RLE , if we assume that the sequences start with "0", these sequences can be encoded as
(1) 2 8 3 2 1
(2) 0 3 2 2 4 3 2
An example of converting a 128x128 bitmap to NoS-RLE format is shown in Figure 8. Assuming that the symbols and run-length values have the same bit-width, then NoS-RLE takes only half as much space as basic run-length encoding.

Algorithm 3 shows our technique for generating NoS-RLE output in context of slicing. In our algorithm, the output is not generated by encoding actual bitmaps. Rather, it is generated using ray-triangle intersections. The new output generation step replaces the layer-extraction step in the original PPS algorithm. During output generation, each intersection trunk obtained from the ray-triangle intersection step is processed by a thread. The intersection trunk is first sorted by the layer number (height) of each intersection, then traversed in order. Each intersection indicates a change in the pixel state, with the initial state being "outside", or "0". Consecutive runs of the same state (due to intersections in adjacent layers) are merged to save space.

Algorithm 3: NoS-RLE Output Generation

```
Input: intersectionTrunk
Output: encodedSequence
/* Sort the intersections in ascending order by z values */
$sortedTrunk \leftarrow$ sort($intersectionTrunk$);
/* Add the first run of pixels outside of the model */
$encodedSequence[0] \leftarrow sortedTrunk[0]$;
$idx \leftarrow 0$;
$prevIdx \leftarrow sortedTrunk[0]$;
while $idx <$ length($sortedTrunk$) do
    /* Find the next run of pixels outside the model */
    /* Consecutive runs of pixels inside the model should be merged */
    $idx \leftarrow idx + 1$;
    while $((sortedTrunk[idx] - sortedTrunk[idx-1] < 1)$ or $(idx\%2 == 1))$ and $(idx <$ length$(sortedTrunk))$ do
        $idx \leftarrow idx + 1$;
    end
    $runOfPixelsInside \leftarrow sortedTrunk[idx-1] - prevIdx + 1$;
    if $idx ==$ length$(sortedTrunk)$ then
        $runOfPixelsOutside \leftarrow numLayers - sortedTrunk[idx-1] - 1$;
    else
        $runOfPixelsOutside \leftarrow sortedTrunk[idx] - sortedTrunk[idx-1] - 1$;
    end
    append $[runOfPixelsInside, runOfPixelsOutside]$ to the end of $encodedSequence$;
end
/* Zero-terminate $encodedSequence$ */
append 0 to the end of $encodedSequence$;
```

Conversion from NoS-RLE to bitmap is similar to decoding an RLE-encoded sequence, except that the symbols now come from the

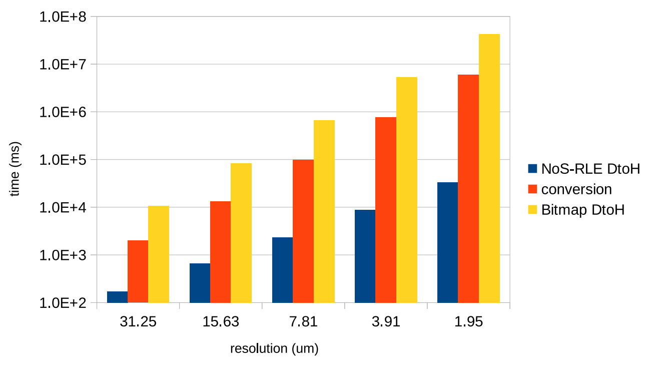

Fig. 9. A comparison between NoS-RLE-to-bitmap conversion time, NoS-RLE DtoH trnasfer time, and bitmap DtoH transfer time.

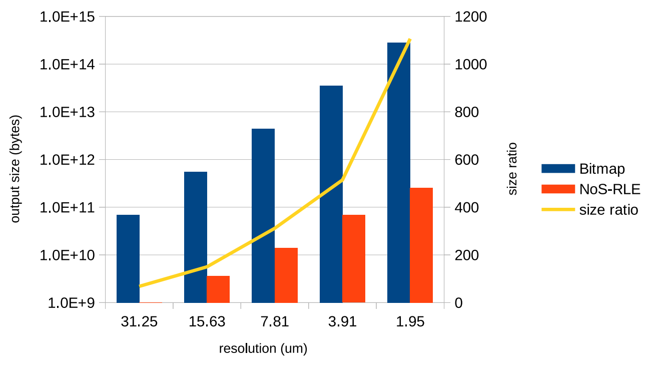

Fig. 11. Output size of model *skull* when using NoS-RLE format. Size ratio is calculated as bitmap size divided by NoS-RLE size for each resolution.

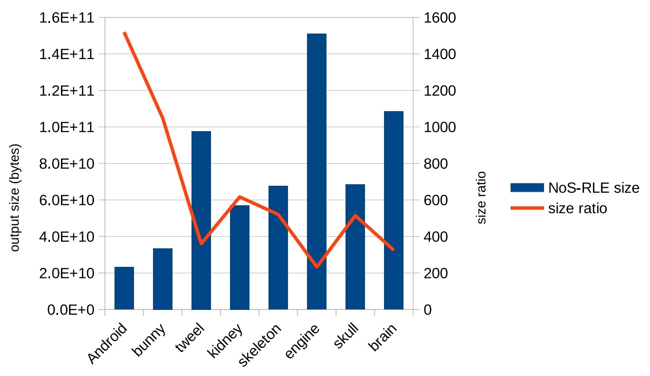

Fig. 10. Output size at 3.91μm when using NoS-RLE format. Size ratio is calculated as bitmap size divided by NoS-RLE size for each model.

internal state of the program rather than the sequence. The internal state starts from “0”, and flips each time a run-length is processed. The overhead of conversion is shown in Figure 9 and compared with the device-to-host transfer time of bitmaps at different resolutions. We can see that even if bitmaps are the required output, it would be much more efficient to generate the bitmaps on the CPU from NoS-RLE format than to copy them from the GPU. At 1.95μm resolution, generating bitmaps from NoS-RLE takes only 1/7th as much time as moving the bitmaps from GPU to CPU. It is also worth noting that the NoS-RLE device-to-host transfer overhead is much smaller compared to the conversion overhead (less than 1% of conversion time at 1.95μm resolution). Therefore, we can expect high speedup if conversion to bitmap is not required.

Because the conversion from NoS-RLE to bitmaps is independent for each encoded sequence, one can either decode multiple sequences in parallel, or pipeline the conversion with whatever task that follows slicing. For example, after the software have finished generating a bitmap, it can send the bitmap to a printer so that the printer prints the current layer while the software starts generating the next bitmap.

Figure 10 shows the size of output in the NoS-RLE format for different models at 3.91μm resolution. We see an average reduction of 642x in size. Although NoS-RLE is expected to be less effective on larger models (since they generally have more ray-triangle intersections), we still see that the average reduction in size is 428x for the large models (i.e., models excluding *Android* and *bunny*). Even for the model with the largest NoS-RLE output size, *engine*, the reduction in size is as high as 233x.

Figure 11 shows the size of output of the model *skull* when sliced at different resolutions. We see that although the size of output in both format increases as the resolution gets smaller, the size of NoS-RLE increases much slower than bitmaps. At 1.95μm resolution, we see up to 1100x reduction in output size.

Figure 12 shows the execution time and speedup of using PPS with NoS-RLE compared to PPS with bitmaps. We can see that the speedup is higher at higher resolution. In addition, we are now able to slice models at resolutions which were not possible when using PPS with bitmaps. With NoS-RLE format, data transfer no longer dominates the execution time even at high resolution. The computation time of PPS has also decreased thanks to the new format, because we can eliminate the cost of layer extraction, which was previously the dominating factor of computation in PPS with bitmaps at high resolution.

In addition to drastically reducing the data-transfer overhead, using NoS-RLE format allows us to eliminate the layer extraction overhead completely, therefore also reducing the computation time. As a result, the total execution time of PPS with NoS-RLE is less than just the computation time of the original PPS. At 3.91μm resolution the total time of PPS with RLE is 5.4×10^2 seconds while the computation time of original PPS is 5.9×10^2 seconds (For 1.95μm the times are 1.7×10^3 and 2.4×10^3 seconds). As a result, if we were to apply compression to the output bitmaps on GPU when using the original PPS algorithm, the overall execution time of PPS with bitmaps (and compression) would still be larger than PPS with NoS-RLE, even if the compression algorithm has zero overhead and infinity compression ratio.

As a point of interest, note that instead of using image-projection based slicing to calculate the output bitmaps directly, one could use contour-based slicing and convert its output (which is equivalent to vector graphics) to bitmaps. A vector graphics based intermediate output may achieve significant reduction in output size compared to a bitmap (similar to what we achieve with NoS-RLE). However, this process has higher computational complexity compared to converting NoS-RLE to bitmaps, because in the output of contour-based slicing, the boundaries of the model are stored as line segments rather than pixels. Besides, we are unaware of any fully functional contour-based slicing algorithms accelerated by GPU. We have seen GPU-accelerated algorithms that generate triangle-plane intersections [37], but none that connects the line segments to make polygons.

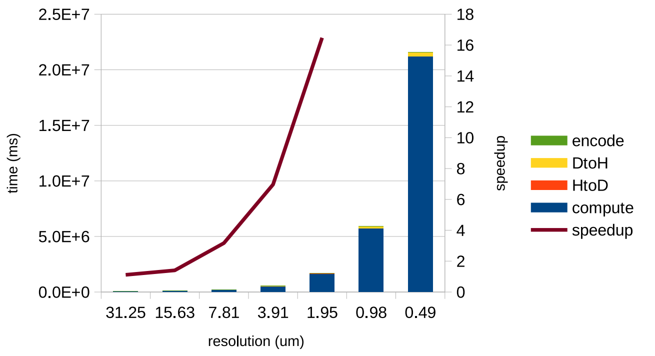

Fig. 12. Execution time of PPS with NoS-RLE as a combination of compute, NoS-RLE encode, DtoH, and HtoD time for model *skull*. The speedup compared to PPS with bitmaps is shown for resolution $\geq$ 1.95µm.

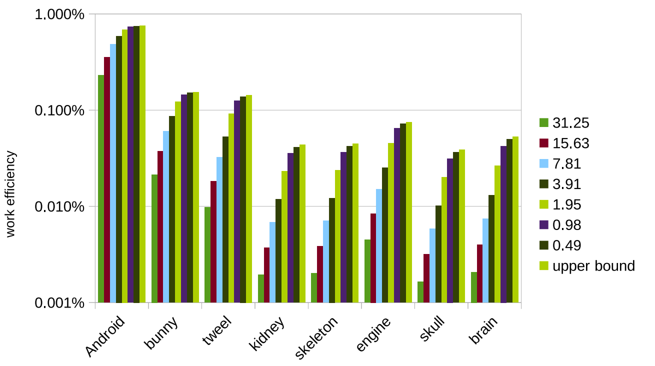

Fig. 13. PPS work efficiency for different models at resolution of 31.25µm-0.49µm.

V. Bounding-Box-Aware Intersection Calculation: A Work-Efficiency Optimization for High-Resolution Slicing

Although the NoS-RLE format greatly decreases the overall execution time, the overall time is still high at sub-micron level (Figure 12). Slicing *skull*, for example, at 0.49µm takes more than 21000 seconds, or about 6 hours. However, unlike PPS with bitmap output where data movement overhead dominated, computation becomes the main overhead this time (since NoS-RLE practically eliminates the data movement overhead). To speed up slicing further, computation overhead of PPS must be reduced.

We define *work efficiency* to be the ratio between the number of valid intersections and the number of ray-triangle intersection calculations performed. In the original PPS algorithm, all pixel-rays are checked for intersections for each triangle, even though only a few pixel-rays actually intersect with the triangle (since the triangle size is usually much smaller than the layer size). Therefore, the PPS algorithm has low work efficiency. From Figure 13 we can see that although the work efficiency increases as we break the model into more blocks at higher resolution, the efficiency has an upper bound. The average efficiency upper bound of the models we have is 0.17%. We also observe that the larger models generally have smaller triangles which leads to lower efficiency (less than 0.05% for the five largest models we have).

We present a high-efficiency slicing algorithm, *Bounding-Box-Aware Intersection Calculation (BBIC)*. First, triangles are randomly put into many different groups. Then, each group is assigned to a GPU thread. Each thread would then iterate over the assigned triangles and calculate all the ray-triangle intersections of each triangle. Unlike in PPS where all pixel-rays on the layer are considered for any given triangle, in BBIC only the pixel-rays within the XY bounding box of the given triangle are considered. Because in BBIC threads are no longer statically mapped to pixel-rays like in PPS, the global memory operations need to be done atomically to avoid data-racing. The pseudo code of this algorithm is presented in Algorithm 4.

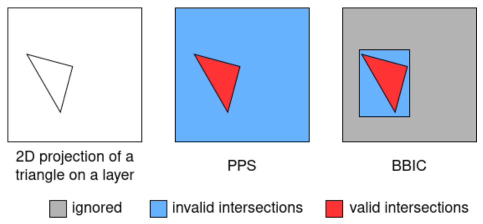

Fig. 14. Difference in intersection-calculation strategies of PPS and BBIC.

Algorithm 4: Bounding-Box-Aware Ray-Triangle Intersection Calculation

```
Input: assignedTriangles
Output: allIntersectionTrunks (shared among all threads)
/* Iterate over all triangles assigned to
   this thread                                  */
foreach tri in assignedTriangles do
    /* Get the XY bounding box of the
       current triangle                         */
    Bbox ← getBoundingBox($tri$);
    /* Iterate over all pixels in the
       bounding box                             */
    foreach (x,y) in Bbox do
        $intersection$ ← computeIntersection($(x, y), tri$);
        if intersection is valid then
            /* Store the z value of
               intersection in the right trunk
               */
            atomically append $intersection$ to
             $allIntersectionTrunks[(x, y)]$;
        end
    end
end
```

The difference between PPS and BBIC is shown in Figure 14. PPS attempts to find an intersection between every pixel-ray and the current triangle, while only the pixel-rays in the red region have valid intersections with the triangle. On the other hand, BBIC ignores all pixel-rays in the grey region and only attempts to find intersections for pixel-rays inside the bounding box (blue and red regions).

With the new slicing algorithm, work efficiency increases by an average of 517x compared to PPS as shown in Figure 15. The highest increase is 948x for *skull*. In general, we observe higher increase in work efficiency when using BBIC on bio-models (*kidney*, *skeleton*, *skull*, and *brain*), likely because they have lower triangle size compared to other models which leads to lower efficiency in PPS.

Figure 16 shows the execution time and speedup of BBIC when slicing the model *skull* at different resolutions. While we see significant speedups from improved work efficiency, the speedup is not as

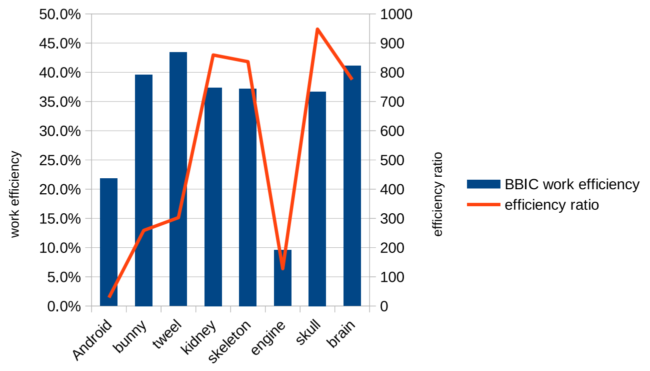

Fig. 15. BBIC work efficiency. Efficiency ratio is calculated as the BBIC work efficiency divided by the efficiency upper bound of PPS for each model.

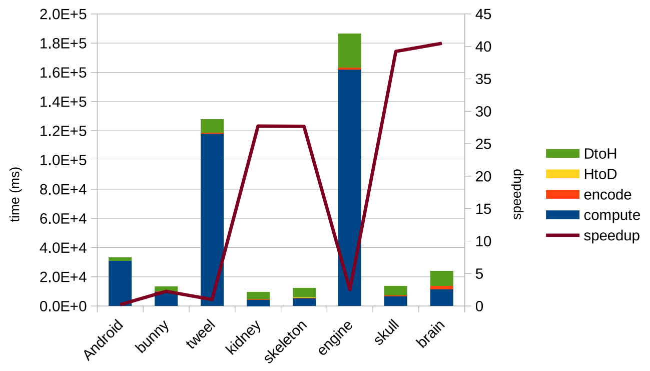

Fig. 17. BBIC execution time and speedup (against PPS with NoS-RLE) for different models at 3.91μm.

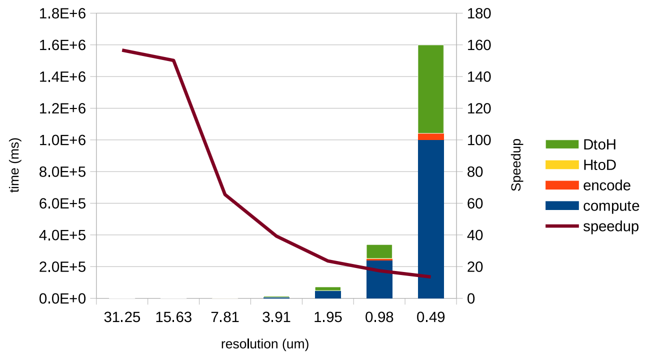

Fig. 16. Execution time of BBIC with NoS-RLE as a combination of compute, encode, HtoD, and DtoH time for model *skull*. Speedup compared to PPS with NoS-RLE is shown for all resolutions.

high as the increase in work efficiency mainly for two reasons: (1) BBIC uses atomic memory operations which have larger overhead compared to non-atomic operations used in PPS; (2) In BBIC warp divergence could be a big problem when triangles assigned to adjacent threads have very different bounding-box size, while in PPS there is barely any warp divergence because all threads are performing the same amount of calculations on the same set of triangles. We see higher speedups at lower resolution because of the poor efficiency of PPS at low resolution. However, the efficiency of PPS saturates as resolution increases. This causes the speedup from BBIC to go down with resolution and saturate. The speedup at highest resolution (0.49μm) is 14x.

We also use BBIC on other models, and the execution time when slicing at 3.91μm is shown in Figure 17. On average BBIC achieves 18x speedup over PPS. We observe that all bio models have over 27x speedup, while the speedup of other models are relatively low, mainly because of the relatively low increase in work efficiency of BBIC over PPS for these models.

When compared to the original PPS with bitmap output format, BBIC with NoS-RLE achieves 388x speedup at 1.95μm resolution for the model *skull* as shown in Figure 18. We can see from the figure that the speedup is higher at higher resolution.

The overall speedup at 3.91μm resolution on all our models is shown in Figure 19. On average, the speedup is 193x. For most of the models, the speedup is above 150x. The speed up is lower for *tweel* and *engine* (28x and 20x respectively), mainly because of their relatively large slicing output size in NoS-RLE format.

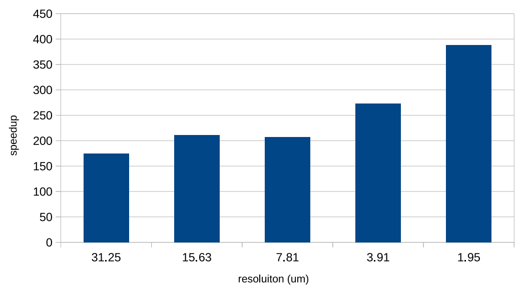

Fig. 18. Overall speedup of BBIC with NoS-RLE compared to PPS with bitmaps when slicing the model *skull* at different resolutions.

VI. Summary and Conclusion

While previous slicing algorithms have been effective at low resolutions, it is unclear if they can be used for fabrication at micro and nanoscale resolutions. We showed that that slicing overhead for existing algorithms may be unacceptable at high resolutions as they produce bitmap-based outputs that may become large at these resolutions (often too large to fit into memory!). The current algorithms also have poor work efficiency. We proposed a new output format for slicing that reduces the size of the sliced output significantly (by 233x-1519x). We also presented a new slicing algorithm that significantly reduces wasted work (by 29x-948x). Together, the optimizations allow slicing at much higher resolutions while providing significant performance speedups (by 20x-378x).

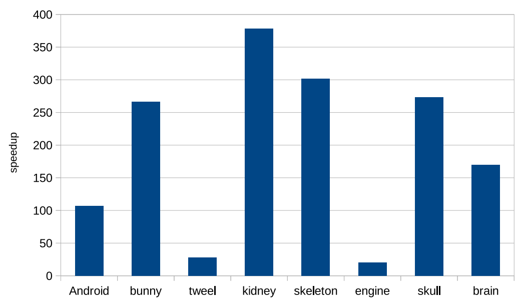

Fig. 19. Overall speedup of BBIC with NoS-RLE compared to PPS with bitmaps when slicing different models at 3.91μm resolution.

References

[1] S. Joshi and A. Sheikh, "3d printing in aerospace and its long-term sustainability," *Virtual and Physical Prototyping*, vol. 10, pp. 1–11, 11 2015.

[2] F. Rengier, A. Mehndiratta, H. Von Tengg-Kobligk, C. Zechmann, R. Unterhinninghofen, H.-U. Kauczor, and F. Giesel, "3d printing based on imaging data: Review of medical applications," *International Journal of Computer Assisted Radiology and Surgery*, vol. 5, pp. 335–341, 07 2010.

[3] C.-Y. Liaw and M. Guvendiren, "Current and emerging applications of 3d printing in medicine," *Biofabrication*, vol. 9, no. 2, p. 024102, jun 2017. [Online]. Available: https://doi.org/10.1088/1758-5090/aa7279

[4] T. Campbell, C. Williams, O. Ivanova, and B. Garrett, "Could 3d printing change the world? technologies, potential, and implications of additive manufacturing," 10 2011.

[5] R. Rael and V. S. Fratello, *Printing Architecture: Innovative Recipes for 3D Printing*. Princeton Architectural Press, 2018.

[6] I. Petrick and T. Simpson, "Point of view: 3d printing disrupts manufacturing: How economies of one create new rules of competition," *Research-Technology Management*, vol. 56, 11 2013.

[7] S. Mueller, S. Im, S. Gurevich, A. Teibrich, L. Pfisterer, F. Guimbretière, and P. Baudisch, "Wireprint: 3d printed previews for fast prototyping," in *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 273–280. [Online]. Available: https://doi.org/10.1145/2642918.2647359

[8] B. Berman, "3-d printing:the new industrial revolution," *Business Horizons*, vol. 55, 03 2012.

[9] R. D. Farahani, M. Dubé, and D. Therriault, "Three-dimensional printing of multifunctional nanocomposites: Manufacturing techniques and applications," *Advanced materials (Deerfield Beach, Fla.)*, vol. 28, no. 28, p. 5794—5821, July 2016.

[10] J. B. Pendry, D. Schurig, and D. R. Smith, "Controlling electromagnetic fields," *Science*, vol. 312, no. 5781, pp. 1780–1782, 2006. [Online]. Available: https://science.sciencemag.org/content/312/5781/1780

[11] A. Nathan, A. Ahnood, M. T. Cole, S. Lee, Y. Suzuki, P. Hiralal, F. Bonaccorso, T. Hasan, L. Garcia-Gancedo, A. Dyadyusha, S. Haque, P. Andrew, S. Hofmann, J. Moultrie, D. Chu, A. J. Flewitt, A. C. Ferrari, M. J. Kelly, J. Robertson, G. A. J. Amaratunga, and W. I. Milne, "Flexible electronics: The next ubiquitous platform," *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1486–1517, 2012.

[12] K. Sun, T.-S. Wei, B. Ahn, J. Y. Seo, S. Dillon, and J. Lewis, "3d printing of interdigitated li-ion microbattery architectures," *Advanced materials (Deerfield Beach, Fla.)*, vol. 25, 09 2013.

[13] H. Lee, E. Koo, M. Yeo, S. Kim, and G. Kim, "Recent cell printing systems for tissue engineering," *International Journal of Bioprinting*, vol. 3, 01 2017.

[14] A. Au, N. Bhattacharjee, L. Horowitz, T. Chang, and A. Folch, "3d-printed microfluidic automation," *Lab Chip*, vol. 15, 02 2015.

[15] H. Lee, "One-step fabrication of an organ-on-a-chip with spatial heterogeneity using a 3d bioprinting technology," *Lab on a chip*, vol. 16, 06 2016.

[16] H. Wang, H. Wang, W. Zhang, and J. Yang, "Toward near-perfect diffractive optical elements via nanoscale 3d printing," *ACS Nano*, vol. XXXX, 07 2020.

[17] J. Y. E. Chan, Q. Ruan, M. Jiang, H. Wang, H. Wang, W. Zhang, C.-W. Qiu, and J. Yang, "High-resolution light field prints by nanoscale 3d printing," 12 2020.

[18] A. Kjar and Y. Huang, "Application of micro-scale 3d printing in pharmaceutics," *Pharmaceutics*, vol. 11, p. 390, 08 2019.

[19] R. Ponni, S. .M, and S. Krishnan, "3d printing in pharmaceutical technology – a review," *International Journal of Pharmaceutical Investigation*, vol. 10, pp. 8–12, 03 2020.

[20] M. Luzuriaga, D. Berry, J. Reagan, R. Smaldone, and J. Gassensmith, "Biodegradable 3d printed polymer microneedles for transdermal drug delivery," *Lab on a Chip*, vol. 18, 03 2018.

[21] T.-Y. Huang, M. Sakar, A. Mao, A. Petruska, F. Qiu, X.-B. Chen, S. Kennedy, D. Mooney, and B. Nelson, "3d printed microtransporters: Compound micromachines for spatiotemporally controlled delivery of therapeutic agents," *Advanced materials (Deerfield Beach, Fla.)*, vol. 27, 09 2015.

[22] X. Ma, J. Liu, W. Zhu, M. Tang, N. Lawrence, C. Yu, M. Gou, and S. Chen, "3d bioprinting of functional tissue models for personalized drug screening and in vitro disease modeling," *Advanced Drug Delivery Reviews*, vol. 132, 06 2018.

[23] "3d printing in the aerospace industry," GRABCAD BLOG, AUG 2019, available: https://blog.grabcad.com/blog/2019/08/27/3d-printing-in-the-aerospace-industry/.

[24] P. Kulkarni, A. Marsan, and D. Dutta, "Review of process planning techniques in layered manufacturing," *Rapid Prototyping Journal*, vol. 6, pp. 18–35, 03 2000.

[25] F. Yang, F. Lin, C. Song, C. Zhou, Z. Jin, and W. Xu, "Pbench: a benchmark suite for characterizing 3d printing prefabrication," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[26] C. Carbon3D, Inc. (Redwood City, "Continuous liquid interphase printing." 2018. [Online]. Available: http://www.library.illinois.edu.proxy2.library.illinois.edu/proxy/go.php?url=http://search.ebscohost.com.proxy2.library.illinois.edu/login.aspx?direct=true&db=edspgr&AN=edspgr.10144181&site=eds-live&scope=site

[27] D. A. Walker, J. L. Hedrick, and C. A. Mirkin, "Rapid, large-volume, thermally controlled 3d printing using a mobile liquid interface," *Science*, vol. 366, no. 6463, pp. 360–364, 2019. [Online]. Available: https://science.sciencemag.org/content/366/6463/360

[28] J. R. Tumbleston, D. Shirvanyants, N. Ermoshkin, R. Janusziewicz, A. R. Johnson, D. Kelly, K. Chen, R. Pinschmidt, J. P. Rolland, A. Ermoshkin, E. T. Samulski, and J. M. DeSimone, "Continuous liquid interface production of 3d objects," *Science*, vol. 347, no. 6228, pp. 1349–1352, 2015. [Online]. Available: https://science.sciencemag.org/content/347/6228/1349

[29] M. Mao, J. He, X. Li, B. Zhang, Q. Lei, Y. Liu, and D. Li, "The emerging frontiers and applications of high-resolution 3d printing," *Micromachines*, vol. 8, no. 4, 2017. [Online]. Available: https://www.mdpi.com/2072-666X/8/4/113

[30] A. Johnson, C. Caudill, J. Tumbleston, C. Bloomquist, K. Moga, A. Ermoshkin, D. Shirvanyants, S. Mecham, J. Luft, and J. DeSimone, "Single-step fabrication of computationally designed microneedles by continuous liquid interface production," *PLOS ONE*, vol. 11, 09 2016.

[31] C. L. Caudill, J. L. Perry, S. Tian, J. C. Luft, and J. M. DeSimone, "Spatially controlled coating of continuous liquid interface production microneedles for transdermal protein delivery," *Journal of Controlled Release*, vol. 284, pp. 122–132, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168365918303250

[32] R. Hai, G. Shao, and C. Sun, "Micro-continuous liquid interface production 3d printing of customized optical components in minutes," 02 2020, p. 43.

[33] "Slash 2 pro," 2020. [Online]. Available: https://www.uniz.com/us_en/3d-printers/slash-2-pro

[34] "Uma 90 technical datasheet," Jun 2020. [Online]. Available: https://3dprinting.com/wp-content/uploads/carbon-uma-90-datasheet-tds.pdf

[35] A. C. Brown and D. de Beer, "Development of a stereolithography (stl) slicing and g-code generation algorithm for an entry level 3-d printer," in *2013 Africon*, 2013, pp. 1–5.

[36] *Layer Depth-Normal Images for Complex Geometries: Part One — Accurate Modeling and Adaptive Sampling*, ser. International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, vol. Volume 3: 28th Computers and Information in Engineering Conference, Parts A and B, 08 2008. [Online]. Available: https://doi.org/10.1115/DETC2008-49432

[37] X. Zhang, G. Xiong, Z. Shen, Y. Zhao, C. Guo, and X. Dong, "A gpu-based parallel slicer for 3d printing," in *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, 2017, pp. 55–60.

[38] A. Wang, C. Zhou, Z. Jin, and W. Xu, "Towards scalable and efficient gpu-enabled slicing acceleration in continuous 3d printing," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 623–628.

[39] "Makerbot thingiverse," available: https://www.thingiverse.com/.

[40] "The biomedical 3d printing community," Embodi3D, available: https://www.embodi3d.com/.

[41] S. Mohamed and M. Fahmy, "Binary image compression using efficient partitioning into rectangular regions," *IEEE Transactions on Communications*, vol. 43, no. 5, pp. 1888–1893, 1995.

[42] S. Zahir and M. Naqvi, "A new rectangular partitioning based lossless binary image compression scheme," in *Canadian Conference on Electrical and Computer Engineering, 2005.*, 2005, pp. 281–285.

[43] "10 fastest 3d printers in the world 2017," APL 2017, available: https://all3dp.com/1/worlds-fastest-3d-printer-speed-3d-printing/.