Fast Prediction of 3D Printing Optimal Orientation  
using Generation Purpose Graphic Card Unit Calculation

Jae Ryoung Kim, Heeran Lee, and In Hwan Sul†

Department of Materials Design Engineering, Kumoh National Institute of Technology, Gumi 39177, Republic of Korea

†: All correspondences to In Hwan Sul ([snowman0@kumoh.ac.kr](mailto:snowman0@kumoh.ac.kr))

# Abstract

This paper proposes a fast parallel computation that can predict the optimal orientation of 3D printing with the aid of general-purpose graphic card unit (GPGPU) calculation. Previously, we proposed the support structure tomography that approximates the amount of support structure as the shadow volume from virtual sunlight, and it showed that the calculation speed was about 1 second per orientation for 65k triangular mesh data. The previous algorithm was converted to a GPU-friendly form using high-and low-priced NVIDIA graphic card and CUDA toolkit. Despite several bottlenecks, calculation speed was improved by two to forty times.

# Keywords

3D printing; optimal orientation; modified support structure tomography; GPGPU; NVIDIA CUDA

# Introduction

Three-dimensional printing (*3DP*), which is also known as rapid prototyping (*RP*) or additive manufacturing (*AM*) [1], is a typical example of the fourthindustrial revolution and is applied to various areas, including mechanical and aerospace [2] engineering, architecture [3, 4], agriculture [5], food industry [6], medical [7], and bio-industry [8, 9]. It is useful when low production volumes, high design complexity, and frequent design changes are required. Its limitations [10] include void formation, anisotropic mechanical properties, and relatively small build volume to CNC machines. The biggest obstacle among them is the support structure (*SS*). Most 3D printers, except for the selective laser sintering (*SLS*) type, require it when the inclination of the polymer stacks is above a certain critical angle. It increases the total printing time and filaments and leads to an unsmooth surface owing to the stair-casing effect [11].

Printer users can use several options to minimize the total number of filaments and surface artifacts. Dai et al. [12] reduced the support structure by rotating the printer bed. However, such a particular device is not feasible for ordinary consumers, and this study assumed that the movement of the bottom plate follows that of typical fused deposition modeling (FDM) printers.

The other approach is to find the optimal orientation in advance theoretically. It can be done either explicitly or implicitly. The explicit method calculates the amount of support structure via direct slicing. To accelerate the calculation, Wang et al. [13] and Das [14] voxelized the target mesh data and reduced the G-code generation time using a graphics processor unit (*GPU*). Their work showed successful g-code and SS information for a specific orientation but not applied to optimal orientation search, presumably due to computation burden [15]. The implicit method indirectly calculates the amount of SS without g-code generation. Ezair et al. [16] observed that the volume of a support structure is similar to that of an object’s shadow volume from virtual sunlight. Their idea was simple and intuitive. However, it applied only to a case of zero filament critical angle. Our team [17] designated this approach as a “shadow-casting analogy” and developed it to a more general form so that any critical angle can be used. We designated the new idea as “modified support structure tomography (*MSST*)” [17, 18], as the resultant distribution of *SS* resembled medical X-ray figures somewhat. The MSST was applied to several well-known meshes, such as Stanford Bunny, and it showed almost the same SS results as conventional slicing software. The only drawback was calculation speed. Our previous method took about one second for a single orientation SS calculation of 65k Bunny. Still, it should be iterated with respect to the X-axis (yaw) and Y-axis (pitch) direction for optimal orientation search. For example, if the search step (***θ****yp*) to the yaw and pitch is 1° respectively, 360 × 360 = 129,600 iterations are needed, and the CPU-based MSST will take tens of minutes for 65k Bunny.

By the way, deep learning technologies such as chatGPT have been developed intensively in the last decade. The main reason would be that the internet provided a massive amount of data, and the hardware, such as NVIDIA’s graphic card unit (*GPU*) or Google’s tensor unit (*TPU*), has been able to deal with the big data efficiently [19, 20]. For example, the learning of *GPU*-based chatGPT-3.5 was 1,000 times faster [21] than the central processing unit (*CPU*), although it had 96 layers and 175 billion parameters [19]. The original role of GPU started with sending pixel color data to the display device. However, the advancement of 3D games boosted the performance of GPUs and vice versa.

Meanwhile, some scientists have started to use GPUs' parallel computing power for general-purpose calculations. NVIDIA, one of the significant GPU companies, opened a C-language-based software development toolkit (SDK) “CUDA” in 2007 [22], and its versatility made it the standard for general-purpose graphic card unit (GPGPU) calculation. However, the bottleneck of GPUs is that their hardware structure still limits them, so conventional CPU-friendly algorithms can only partially utilize the GPU's parallel computing power.

This paper converted our previous MSST algorithm [23] to a GPU-friendly one to predict the optimal orientation of a complex 3D mesh object quickly and in a reasonable time. Chapter 2 briefly reviews our previous MSST algorithm and explains the new GPU version. Chapter 3 introduces the hardware and mesh data used. Chapter 4 compares and discusses the performance between CPUs and GPUs.

# Modeling

## Brief review of our previous work [17, 18, 24]

### Modified Support Structure Tomography

Figure 1 illustrates Ezair et al.’s idea [16]. The target object to be printed is laid on the printer bed (Figure 1a). Virtual sunlight is projected from the far +Z axis direction, and the objects' top-covering (TC) surfaces make a shadowed volume (Figure 1b). As the object’s volume (***V****o*) is constant (Figure 1c), the volume of SS (***V****ss*, Figure 1d) can be known by Eq. (1).

(1)

, where the graphic card’s depth test can easily give the top-covering volume (***V****tc*).

|  |  |
| --- | --- |
| (a) | (b) |
| (c) | (d) |

Figure 1**.** Ezair et al. [16]’s implicit support structure calculation scheme (a) target object, (b) Calculating top-cover volume using GPU, (c) calculating object volume, and (d) final support structure volume (=*Vtc*-*Vo*) [25]

By the way, Eq. (1) is applicable only when the filament critical angle (***θ****c*) is zero, which means that every overhang needs a support structure, as shown in Figure 1d. Most 3DP uses polymeric filaments, and they have ***θ****c* value between 45°~60°, which means that some overhang does not need *SS*. Thus, the more general form of Eq. (1) will be Eq. (2)

(2)

, where ***V****nv* represents the non-overhang volume.

### Pixel-slot-based volume integration

Calculating ***V****nv* is the tricky part of this algorithm. The occlusion between the overhang surface and the bottom plate made *SS* volume in Figure 1d. However, most of the real-life objects have more complex surface-to-surface occlusion. To deal with this, we split the virtual sunlight of Figure 1 into multiple rays. Figure 2 illustrates this approach. Each ray meets the object's surface, and we designate the intersection points as “pixels.” Figure 2a shows four pixels from the same ray. We sort these pixels by the z-coordinates and designate them as “pixel slots.” Then, the pixels facing the virtual sun direction (+Z) are called α pixels, and those facing the bottom plate are *β* pixels. Interestingly, integrating the z-components of those pixels can give object volume like Eq. (3) [17].

(3)

And then inserting Eq.(3) to Eq.(2) gives the Eq. (4)

(4)

Also, we can easily find a TC pixel equivalent to the highest *α* pixel in the slot (*α0* in Figure 2a). It is also added to the current pixel group and the bottom intersection point. And then, surface-to-surface occlusion intervals are found, such as (*β0*, *α1*) and (*β1*, *bottom*). These intervals are candidates for either SS or NV. For example, all the intervals will be *SS* if ***θ****c* is zero (Figure 2b). Otherwise, some will be *NV*, depending on the overhang’s inclination angle (Figure 2c).

|  |  |  |
| --- | --- | --- |
| (a) after  pixelization | (b) *θ*c=0° | (c) *θ*c=60° |
|  | | |

Figure 2. Classification of pixel type in a pixel slot (x,y). [18]

Repeating the procedure of Figure 2c for every area of the target object can give the filament volume information such as ***V****α*, ***V****β*, ***V****tc*, ***V****o*, and ***V****ss* for a specific orientation. Figure 3 is the schematic overview of the *MSST* algorithm for a simple boat-like shape.



Figure 3**.** Schematic overview of our modified support structure tomography method for a simple boat-like shape [18]

### Volume-to-mass conversion

3D Printers do not fill the inside volume for printing speed. We dissected the object volume (***V****o*) into external shells (“clad”) and internal (“core”), like Eq. (5) and Eq.(6) [24].

(5)

(6)

With I and J representing the number of α and β pixels and Tclad denoting the thickness of the clad, the value of which is derived from the g-code options, the volume-to-mass conversion process is straightforward and uncomplicated, as demonstrated in Eq. (7)-(12). This simplicity should instill confidence in its application.

(7)

(8)

(9)

(10)

(11)

(12)

, where ***Fclad***, ***Fcore***, and ***ρPLA***represent the clad fill ratio, core fill ratio, and filament density [24].

### Orientation contour graph

To find the optimal printing orientation, we need to repeat the procedure of 2.1.2, as shown in Figure 4. The right-hand-side contour graph is the total filament mass (***M****total*) for (yaw, pitch). The search step (***θ****yp*) less than 15° did not make much difference in ordinary shapes in our previous works, but 1° was used in this paper. The bluest regions mean the optimal orientations in the contour graph. Texts “o#” and “w#” represent #st optimal and worst.



Figure 4**.** Flowchart of optimal orientation search using modified support structure tomography

## GPU-oriented modification

### Subdivision of triangles

The first step in the GPGPU-version MSST algorithm is to voxelize input mesh data. We split the voxelization into two steps. The CPU subdivides each triangle into four pieces iteratively until its fragments fit 4 mm x 4 mm, and then the GPU voxelizes the fragments, as shown in Figure 4.



Figure 5**.** Illustration ofGPU-friendlyvoxelization scheme usingtriangular subdivision

### Slot-based pixel data structure

Due to the limitations **i)** ~ **v)**, storing an unknown number of data in GPU is not convenient. Therefore, we assumed that the 3D printer’s capacity is 256×256×256 mm3. Figure 5a shows a bird’s-eye view of the pixel slots. Figure 5b represents the detailed view of each pixel group data structure. Considering the GPU warp size (limitation i)), we designed each pixel group to have 3 x 16 bytes, respectively. Each three-byte triplet contains minimal pixel data such as pixel type, z-coordinate, and normal vector’s z-component. Only the first triple was reserved for slot meta-data, such as the number of pixels, local volume, and local support structure volume. In this way, an empty slot can be shown in Figure 5c. Figures 5d and 5e illustrate an exemplary slot before and after sorting by the z-coordinate.

|  |
| --- |
| (a) bird’s-eye view of pixel slots |
| (b) pixel data format in each slot |
| (c) example of an empty slot |
| (d) example of an unsorted slot with two pixels |
| (e) sorted result of (d) |

Figure 6**.** Illustration of slot-based pixel data structure (nV: number of pixels, Vo: object volume, Vss: support structure volume, Tp: pixel type(*α,β, TC, NV, SS, o*), z: pixel Z-coordinate, nZ: normal vector’s Z-component)

# Experimental

## 3D printer

A FDM-type dual-nozzle 3D printer (Sindoh 2X DP-303®, Korea) was used as the target printer. The bundle slicer software 3DWOX®, which internally uses Ultimaker CuraEngine® for slicing, was used for G-code generation. A PLA filament was chosen because it is widely used and environmentally friendly. The printer had a maximal printing volume of 228 × 200 × 300 mm3, and we set the voxel size (*dVoxel*) and number (*nVoxel*) to 1 mm and 2563, respectively, assuming that the maximal length of the input object was 256 mm. Table 1 shows the default G-code options of the slicing software, and the same values were also used in our modeling. Parameter names in parentheses indicate those used in the *MSST*. Some of the g-code options were changed in filling density (100%), support structure (everywhere, 100% density, critical angle 1°), bed filling(none), and internal filling (linear type). The other options were unchanged. The PLA filament’s density (1.121 g/㎤) was measured by Korea Polymer Testing & Research Institute Ltd. with ASTM E1461 [26].

Table 1. Specification of the “default” g-code generation options for the slicing software.

|  |  |  |
| --- | --- | --- |
| category | option | value |
| Filament | material | PLA |
| diameter | 1.75 mm |
| density (***ρ****PLA*) | 0.0121 g/mm3 |
| layer height | 0.20 mm |
| Wall  (“clad”) | thickness (***T****clad*) | 0.8 mm |
| infill ratio (***F****clad*) | 100% |
| Infill  (“core”) | infill ratio (***F****core*) | 15% |
| fill pattern | linear |
| Support structure (“SS”) | critical angle (***θ***c) | 60° |
| infill ratio(***F****ss*) | 20% |
| fill pattern | linear |
| line width (***C****ss*) | 100% |
| horizontal expansion (***C****ss*) | 0.8 mm |
| Bedplate | type | raft |

## Mesh data

Table 2 shows the test mesh data and their specifications. Cube, Sphere, and Cone were drawn using AutoDesk TinkerCAD®. The Stanford mesh data series were downloaded from the Stanford 3D Scanning Repository[31], and their size and mesh resolution were adjusted. The manikin data (“Masha,” 6,690 vertices, 13,672 triangles) were obtained in OBJ format and recalled to 1/10.

Table 2. Specification of the test mesh data

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Name | #Vtx/#Tri | dimension  [mm^3] |
|
| (1) | cube | 8./12 | 50x50x50 |
| (2) | sphere | 1,502/3,000 | 75x75x75 |
| (3) | cone | 3,152/6,300 | 100x100x100 |
| (4) | B\_2x | 34,834/69,662 | 151x117x150 |
| (5) | B | 75x58x75 |
| (6) | B\_0.5x | 38x30x37 |
| (7) | B/5k | 2,502/5,000 | 75x58x74 |
| (8) | B/1k | 502/1,000 | 76x58x74 |
| (9) | mnk | 6,882/13,672 | 135x33x174 |
| (10) | dragon | 50,062/99,999 | 167x75x118 |
| (11) | Buddha | 24,943/49,944 | 52x52x128 |
| (12) | Lucy | 25,006/49,999 | 104x60x180 |

## Computation

C++ compiler (Visual Studio 2019 Community Edition) was used for the CPU version, and CUDA toolkit 12.1 was used for the GPU version on MS Windows 11 PC. Results were rendered using Python (V3.7.5) with *Numpy*, *Matplotlib*, *Plotly*, and *Open3d* packages. CPUs/GPUs with various prices were chosen, and their specifications are shown in Table 3 and Table 4. NVIDIA Visual Profiler measured CUDA’s speed. CUDA’s *cudaMemcpyAsync* function was used for CPU-GPU data transfer, and the warp-reduction function [32] was used to integrate the pixel slot’s resultant volume values.

Table 3. Specification of the CPUs used

|  |  |  |
| --- | --- | --- |
| Name | Base/max clock  (GHz) | # of core/ thread |
| Intel i7-8700 | 3.2/4.6 | 6/12 |
| AMD Ryzen9 7950X | 4.2/5.7 | 16/32 |

Table 4. Specification of the GPUs used

|  |  |  |
| --- | --- | --- |
| Name | Base/boost clock(GHz) | # of stream processor(SM) |
| NVIDIA GTX760 | 0.98/1.03 | 1,152 |
| GTX1050Ti | 1.29/1.39 | 768 |
| RTX 2060 | 1.37/1.68 | 1,920 |
| RTX 3080 Ti | 1.37/1.67 | 10,240 |
| RTX 4090 | 2.2/2.5 | 16,384 |

# Results and Discussion

## Estimation for specific orientation

|  |  |
| --- | --- |
| CPU 결과 그림 추가!!   1. CPU version (***M***ss= ?? g) | (b) GPGPU version (***M***ss=9.2 g) |

Figure 7**.** voxelization result for *Bunny 69k 2x* at (0°,0°) orientation (red: support structure, blue: bottom structure)

앗,, CPU/GPU 결과값이 살짝 다른걸.. 오차 비교를 해야 하나.. Slicing SW vs CPU vs GPU?

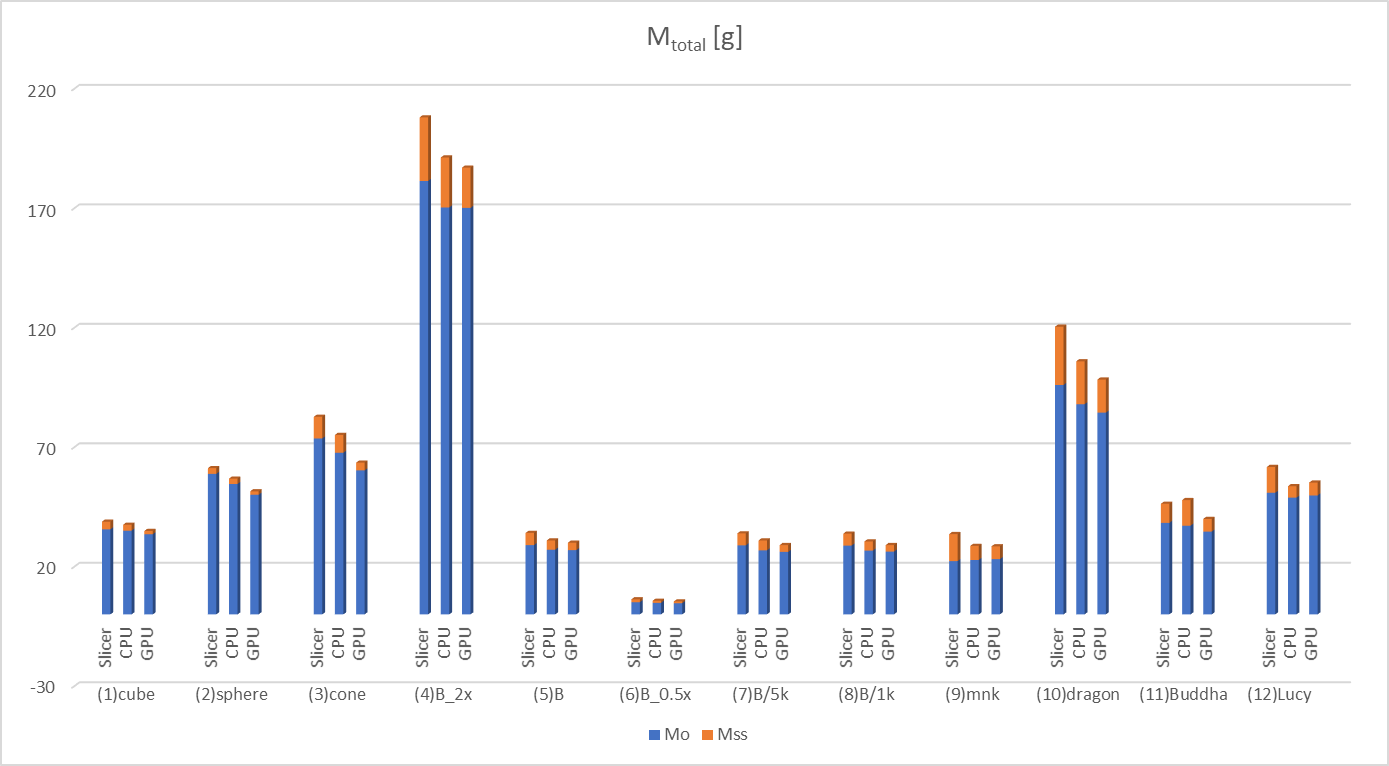


그림 3은 2.1절에서 설명한 지지구조 단층촬영법을 GPU화하여 (0°,0°) 배향의 Bunny 데이터에 대해 지지구조량을 분석한 사례이다. 빨간색 선분이 예상되는 지지구조-의 위치를 나타내고 있다. 이러한 특정 배향에 대한 계산은 CPU기준 1초 미만의 매우 빠른 속도로 계산이 가능하다. …..계산 결과에 거의 차이가 없음을 보아 GPGPU 코드 변환이 잘 이루어진 것으로 보인다.

### Obstacles in GPGPU Programming

Our concern is to modify the *MSST* algorithm to a GPU-friendly version. Unfortunately, some GPGPU applications show slight improvement because of the following limitations [27];

* **i)** Data vectorization needed

Data stored in a series, such as a picture or a movie file, is said to be ‘vectorized.’ CPU prefers vectorized data, but it is not a necessary condition. On the contrary, GPU assumes that all the input and output data are vectorized. It has been a hot issue for the last decade to deal with sparse data, which is common in scientific problems. These results are helpful for commonly used algorithms such as matrix multiplication [28]. However, GPGPU users must develop a new vectorized data structure for unique problems such as *MSST*.

* **ii)** Fixed warp size

In fiber engineering, a sewing yarn is composed of several fibers. In the same way, modern GPUs execute 32 threads simultaneously with the same shader program to improve computation density [28]. These 32-thread groups are designated as “warp” by NVIDIA and share a common fast cache memory, “shared memory.” Therefore, a GPGPU user should prepare a data size multiple of 32 to achieve a maximal GPU calculation speed.

* **iii)** Control flow divergence problem

The other advantage of using a multiple of 32 threads is that all the threads are fully operational. However, the user’s algorithm may need conditional statements such as ‘if-then.’ In that case, some threads of a warp work while the others rest, which results in a performance drop. This is called the “control flow divergence” problem [27].

* **iv)** Gather-scatter operation

GPU shaders did not need to communicate with their neighbors for their original purpose, i.e., rendering colors on the monitor. However, it is expected to access neighbors' data in GPGPU problems; i.e., sometimes neighboring shaders’ data should be summed, or a shader’s data should be transferred to its neighbors. These are designated as “gather” and “scatter” operations, respectively, in GPGPU programming [29]. These were difficult jobs in the early versions of GPGPU, such as OpenGL 2.0. However, they became more accessible with the help of GPU providers' new functions, such as parallel reduction with shared memory [30]. The programmer should still carefully design the shader algorithm and data structure.

* **v)** CPU–GPU memory transfer overhead

Semiconductors such as CPU, GPU, and DRAM can execute operations quickly. However, the motherboard’s PCIe interface, which connects them, has much slower bandwidth [30]. Therefore, it is advantageous to minimize CPU–GPU memory transfer or to transfer the data asynchronously while the GPU is not busy.

## earching for optimal orientation

러나 최적 배향을 찾기 위해서는 4.1의 계산을 반복해야 한다. Figure 8a,b는 각각 CPU와 GPU로 1도 간격으로 배향을 달리하며 지지구조 질량을 측정한 결과를 등고선 형태로 나타낸 것이다. 파란색이 가장 지지구조 양이 작은 최저 배향을 나타낸다.

|  |  |
| --- | --- |
| (a) orientation graph (left: CPU, right: RTX 4090 GPU, angle interval:1°) | |
| (b) 1st Optimal orientation (231°,54°), ***M***ss=4.8g | (c) 1st worst orientation (243°,356°), ***M***ss=63.4 g |

Figure 8. Optimal orientation search example for *Bunny 65k 2x*

표 2은 계산 시간을 측정한 결과로서, 본 실험에 사용한 CPU의 경우 6개의 코어가 동시에 12개의 쓰레드로 각도별 연산을 분산 처리할 수 있다. 예를 들어 첫번째 쓰레드는 (0°, 0°) 배향일 때의 모든 삼각형의 복셀화 및 필라멘트 질량 계산을 모두 도맡아 처리하고, 두번째 쓰레드는 (0°, 1°) 배향의 계산을 모두 처리하는 방식이다. 사용된 6-코어 CPU의 경우 총 14.29초의 시간이 소요되었는데, CPU의 쓰레드 개수가 많을 경우 시간은 더 단축될 수도 있다. 마찬가지로 GPU의 경우도 1152개의 코어가 동시에 동작하고, 이때 사용할 스트림의 개수를 지정할 수 있는데, 1에서 10까지 스트림 개수를 달리하며 측정한 결과 Table 2과 같이 오히려 CPU보다 느린 속도가 측정되었다. 첫번째 이유는 데이터의 이산화이다(scattered data). 일반적인 딥러닝은 데이터의 일렬로 존재하므로 병렬 계산에 적합하지만, , 우리 데이터는 Figure 5b와 같이 픽셀 슬롯에 들어있는 데이터들이 일정하지 않다. 두번째 원인은 알고리즘에 있다. GPU는 모든 코어가 동시에 같은 일을 할 때에만 최대 성능을 발휘한다. 특히 if문과 같은 조건문이 사용될 경우 유휴 코어가 생기므로 성능이 급감하게 된다. 그럼에도 불구하고 본 알고리즘의 GPGPU버전은 Table 2에서와 같이 저가형 그래픽카드인 GTX760에서도 CPU와 비슷하거나 더 빠른 성능을 나타내었다.

Table 5. Calculation time of CPU and low-priced GPU for Stanford Bunny with respect to different max triangle sizes (angle interval: 10°, unit: [minute:second])

|  |  |  |  |
| --- | --- | --- | --- |
| Hardware | | Max triangle size (width x height) | |
| type† | # of thread | 16x16 | 4x4 |
| CPU | 12 | 0:14.3 | |
| GPU | 1 | 0:23.3 | 0:09.3 |
| 2 | 0:21.0 | 0:07.7 |
| 3 | 0:22.6 | 0:08.9 |
| 6 | 0:28.3 | 0:11.6 |
| 10 | 1:06.3 | 0:28.5 |

(†: CPU: Intel i7-8700, GPU: NVIDIA GTX 760)

세부함수별 걸리는 시간을 분석하기 위해, NVIDIA가 제공하는 GPU 성능 분석 툴인 Visual Profiler를 이용하였다. Figure 9와 Table 3은 GTX760으로 Bunny 69k 데이터를 1도 간격으로 최적 배향을 계산할 때의 화면이다. 스트림 15~18이라는 4개의 thread를 사용했을 때, 각 쓰레드가 순차적으로 동작하고 있다. 이는 GPU의 성능이 부족하여 전체 동작을 병렬화할 여유가 없기 때문이다. 세부적으로 볼 때 가장 시간이 많이 소요된 부분은 바닥구조 계산 함수 (cu\_genBed)였고, 이후 복셀 회전(cu\_rotVoxel), 픽셀 정렬 및 노이즈 제거 (cu\_slotPairing), 각 슬롯의 부피값 합산 (cu\_reducedSum)의 순이었다.

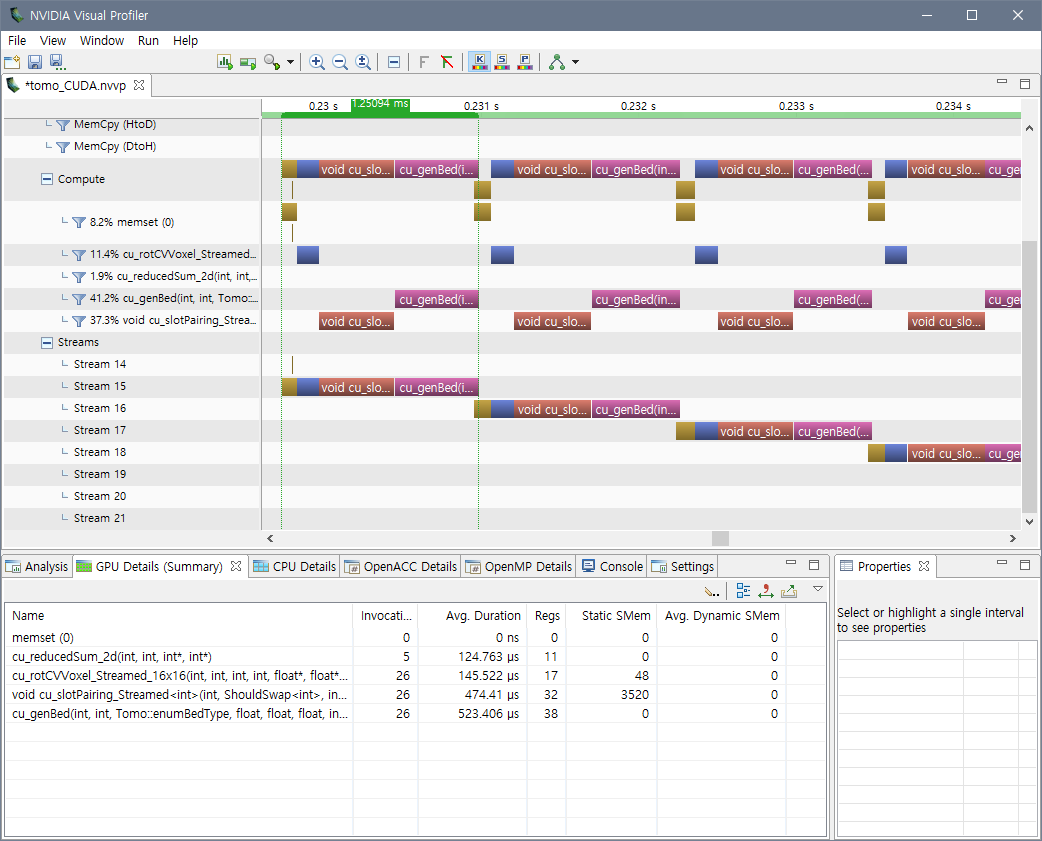


Figure 9**.** NVIDIA Visual Profiler screenshot (GPU: GTX 760, mesh: Bunny 69k)

Table 6. Calculation time of CUDA functions at (0°,0°) orientation (GPU: RTX 4090, mesh: Bunny 69k)

|  |  |  |
| --- | --- | --- |
| function | Grid x block size | Time [㎲] |
| cu\_genBed() | (256) x (256) | 35.1 (38.3%) |
| cu\_rotVoxel() | (3298) x (4,4,32) | 33.3 (30.2%) |
| cu\_slotPairing() | (4096) x (16,16) | 28.5 (27.6%) |
| cu\_reducedSum() | (256) x (256) | 4.4 (4.0%) |
| Σ | - | 101.3 (100.0%) |

Table 4는 cone, sphere, Bunny, Manikin등 다양한 메쉬 형상에 대한 최적 배향 계산 결과이다. CPU는 인텔 i7-8700을, GPU는 저가형인 NVIDIA GTX 760을 이용했으며, 속도 개선은 30배 내지 66배로 나타났다.

Table 7. Calculation time comparison for optimal orientation search (angle interval: 1°, unit: [minute:second])

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Time  Test data | | | Calculation time‡ | |
| ID | #Vtx/#Tri | dimension  [mm^3] | CPU | GPU (speedup) |
| Cone | 3,152/6,300 | 100x100x100 | 7:13.8 | 0:12.7 (x34.2) |
| Sphere | 19,602/39,200 | 180x180x180 | 13:15.1 | 0:21.2 (x37.5) |
| Bunny 69k 2x | 34,834/69,662 | 151x117x150 | 14:07.4 | 0:12.7 (x66.7) |
| Bunny 69k | 75x58x75 | 12:06.3 | 0:11.7 (x62.1) |
| Bunny 69k 0.5x | 38x30x37 | 11:05.8 | 0:10.9 (x61.1) |
| Bunny 5k | 2,502/5,000 | 75x58x74 | 7:03.2 | 0:10.0 (x42.3) |
| Bunny 1k | 502/1,000 | 76x58x74 | 6:26.1 | 0:10.0 (x38.6) |
| Manikin | 6,882/13,672 | 33x135x174 | 7:23.0 | 0:10.2 (x43.4) |

(‡: CPU: Intel i7-8700, GPU: NVIDIA GTX 760)

Figure 10는 고가형 그래픽카드인 NVIDIA RTX 4090을 이용하였을 때의 성능 분석 화면이다. GTX760과는 달리 그래픽카드의 성능이 충분하여 일부 스트림들은 동시에 병렬로 동작하고 있음을 보여준다. Figure 11은 다양한 그래픽카드별 성능을 나타낸 것이다. 각각 3번씩 측정하였으나, 시간 차이가 거의 없으므로 에러바는 생략하였다. Table 1에 나타난 그래픽 카드의 코어 개수에 비례하여 연산 속도가 향상되는 것을 확인할 수 있다.

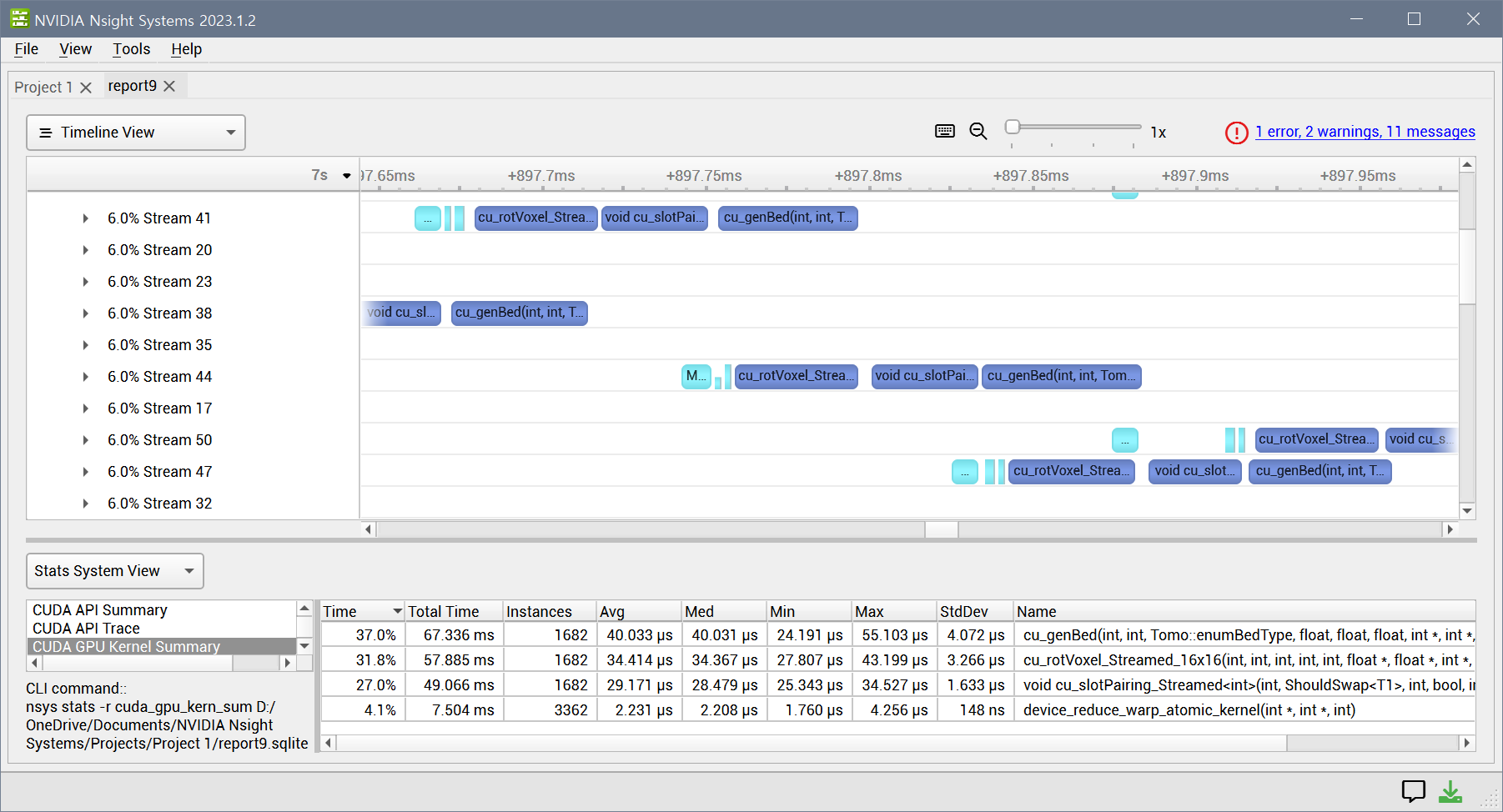


Figure 10**.** NVIDIA Nsight systems screenshot (GPU: GTX 4090, mesh: *Bunny 69k*)

Figure 11**.** Calculation results of various CPU/GPU(mesh: *Bunny 69k*)

*GPGPU 연산은 딥러닝 등 벡터 데이터의 계산에 널리 활용되고 있다. 반면 본 연구에서 사용한 일반 메쉬 데이터는 GPGPU연산에 적합한 형태는 아니지만, 투고일 기준 5만원 대의 저가형 그래픽 카드를 이용하더라도 2배의 연산 속도 향상이 이루어졌다. 따라서 스트림 프로세서 개수가 더 많은 고가의 그래픽카드를 이용할 경우 추가적인 속도 개선이 있을 것으로 예상된다. 또한 이러한 연산 기술은 향후 효율적인 딥러닝 함수 개발에도 활용될 수 있을 것이다.*

# 결론

이전 연구에서 개발한 복셀화 기반 지지구조량 예측 기법인 일명 단층촬영법을 그래픽카드를 이용하여 GPGPU연산화하였다. 저가형 그래픽카드의 경우 2배, 고가형의 경우 60배 정도의 속도 향상이 측정되었다. 원인이 된 제약사항으로는 CPU-GPU간 데이터 전송 속도 한계, 벡터화된 데이터에만 적합한 GPU 내부 구조, GPU 내부 스트림간 데이터 공유의 어려움, GPU 소스코드에서의 조건 문 사용으로 인한 효율 저하, GPU 와프 사이즈 제한으로 인한 성능 저하 등이 있었다. GPU 스트림의 개수는 사용자가 지정할 수 있지만, 2개가 적당하였으며 더 많은 수는 오히려 효율이 떨어졌다. 삼각형의 크기의 경우 와프 사이즈인 16개 이하의 복셀 개수가 형성되도록 최대한 작은 크기로 준비하는 편이 가장 연산 효율이 좋게 측정되었다. GPGPU연산을 적용 결과, 지지구조 계산량 측정에는 크게 도움이 되지 못하는 것으로 나타났다. 그러나 이러한 고속 연산 기법은 향후 섬유의류분야 딥러닝 등 벡터화된 데이터의 연구에 있어서는 도움이 될 것으로 기대된다.

**Acknowledgment**: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2022R1A2C1010072 and NRF-2023R1A2C2005376).

# Appendix

Figure A1, A2는 각각 동일한 (x,y)위치에 있는 복셀들을 z좌표를 기준으로 분류(sort)하는 소스코드의 일부 사례이다.



**Figure A1.** CPU source code for pixel vertical sorting algorithm using C++ 11



**Figure A2.** GPU source code for pixel vertical sorting algorithm using NVIDIA CUDA toolkit

# References

1. A. ISO, "ISO/ASTM 52900: 2015 Additive Manufacturing–General Principles–Terminology", in *ASTM F2792-10e1*. 2015.

2. S. C. Joshi and A. A. Sheikh, "3D printing in aerospace and its long-term sustainability"*.* *Virtual and Physical Prototyping*, 2015. **10**, 175-185.

3. F. Bos, R. Wolfs, Z. Ahmed, and T. Salet, "Additive manufacturing of concrete in construction: potentials and challenges of 3D concrete printing"*.* *Virtual and Physical Prototyping*, 2016. **11**, 209-225.

4. A. Perrot, D. Rangeard, and A. Pierre, "Structural built-up of cement-based materials used for 3D-printing extrusion techniques"*.* *Mater. Struct.*, 2016. **49**, 1213-1220.

5. J. Pearce, "Applications of open source 3-D printing on small farms"*.* *Organic Farming*, 2013. **1**, 19-35.

6. C. Guo, M. Zhang, and B. Bhandari, "Model building and slicing in food 3D printing processes: a review"*.* *Comprehensive Reviews in Food Science and Food Safety*, 2019. **18**, 1052-1069.

7. I. Badash, K. Burtt, C. A. Solorzano, and J. N. Carey, "Innovations in surgery simulation: a review of past, current and future techniques"*.* *Ann. Transl. Med.*, 2016. **4**.

8. S. V. Murphy and A. Atala, "3D bioprinting of tissues and organs"*.* *Nat. Biotechnol.*, 2014. **32**, 773.

9. F. Rengier, A. Mehndiratta, H. Von Tengg-Kobligk, C. M. Zechmann, R. Unterhinninghofen, H.-U. Kauczor, and F. L. Giesel, "3D printing based on imaging data: review of medical applications"*.* *Int. J. Comput. Assist. Radiol. Surg.*, 2010. **5**, 335-341.

10. T. D. Ngo, A. Kashani, G. Imbalzano, K. T. Nguyen, and D. Hui, "Additive manufacturing (3D printing): A review of materials, methods, applications and challenges"*.* *Composites Part B: Engineering*, 2018. **143**, 172-196.

11. P. Delfs, M. Tows, and H.-J. Schmid, "Optimized build orientation of additive manufactured parts for improved surface quality and build time"*.* *Additive Manufacturing*, 2016. **12**, 314-320.

12. C. Dai, C. C. Wang, C. Wu, S. Lefebvre, G. Fang, and Y.-J. Liu, "Support-free volume printing by multi-axis motion"*.* *ACM Trans. Graph.*, 2018. **37**, 1-14.

13. C. C. Wang, Y.-S. Leung, and Y. Chen, "Solid modeling of polyhedral objects by layered depth-normal images on the GPU"*.* *Computer-Aided Design*, 2010. **42**, 535-544.

14. P. Das, K. Mhapsekar, S. Chowdhury, R. Samant, and S. Anand, "Selection of build orientation for optimal support structures and minimum part errors in additive manufacturing"*.* *Computer-Aided Design and Applications*, 2017. **14**, 1-13.

15. Q. Xu, H. Jeon, and M. Annavaram. "Graph processing on GPUs: Where are the bottlenecks?". 2014 IEEE International Symposium on Workload Characterization (IISWC), pp.140-149: IEEE, 2014.

16. B. Ezair, F. Massarwi, and G. Elber, "Orientation analysis of 3D objects toward minimal support volume in 3D-printing"*.* *Comput. Graph.*, 2015. **51**, 117-124.

17. J. Y. Jung, S. Chee, and I. H. Sul, "Support structure tomography using per-pixel signed shadow casting in human manikin 3D printing"*.* *Fash. Text.*, 2022. **9**, 1-18.

18. J. Y. Jung, S. Chee, and I. Sul, "Prediction of optimal 3D printing orientation using vertically sparse voxelization and modified support structure tomography"*.* *Int. J. Cloth. Sci. Tech.*, 2023. **35**, 799-832.

19. T. Wu, S. He, J. Liu, S. Sun, K. Liu, Q.-L. Han, and Y. Tang, "A brief overview of ChatGPT: The history, status quo and potential future development"*.* *IEEE/CAA Journal of Automatica Sinica*, 2023. **10**, 1122-1136.

20. C. Zhang, C. Zhang, S. Zheng, Y. Qiao, C. Li, M. Zhang, S. K. Dam, C. M. Thwal, Y. L. Tun, and L. L. Huy, "A complete survey on generative ai (aigc): Is chatgpt from gpt-4 to gpt-5 all you need?"*.* *arXiv preprint arXiv:2303.11717*, 2023.

21. E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr. "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC". 2016 International Conference on Field-Programmable Technology (FPT), pp.77-84: IEEE, 2016.

22. I. Buck, "Gpu computing with nvidia cuda", ACM SIGGRAPH 2007 courses, 2007. Vol., pp.6-es.

23. J. Y. Jung, S. Chee, and I. H. Sul, "Prediction of optimal 3D printing orientation using vertically sparse voxelization and modified support structure tomography"*.* *Internationl Journal of Clothing Science and Technology*, 2023. **accepted**.

24. J. Y. Jung and I. H. Sul, "Prediction of 3D Printing Bottom Plate Amount Using Open Source Software and Support Structure Tomography"*.* *Text. Sci. Eng.*, 2023. **60**, 249-255.

25. J. Y. Jung and I. H. Sul, "Prediction of Filament Usage in Human Manikin 3D Printing Using Voxel Approximation-based Shadow Projection"*.* *Text. Sci. Eng.*, 2022. **59**, 79-87.

26. J. A. Slotwinski and S. Moylan, "Applicability of existing materials testing standards for additive manufacturing materials", US Department of Commerce, National Institute of Standards and Technology …, 2014.

27. M. Khairy, A. G. Wassal, and M. Zahran, "A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity"*.* *Journal of Parallel and Distributed Computing*, 2019. **127**, 65-88.

28. S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs"*.* *ACM Transactions on Mathematical Software (TOMS)*, 2017. **43**, 1-49.

29. B. He, N. K. Govindaraju, Q. Luo, and B. Smith. "Efficient gather and scatter operations on graphics processors". Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, pp.1-12, 2007.

30. M. Harris and M. Garland, "Optimizing parallel prefix operations for the Fermi architecture", GPU Computing Gems Jade Edition, Elsevier, 2012. Vol., pp.29-38.