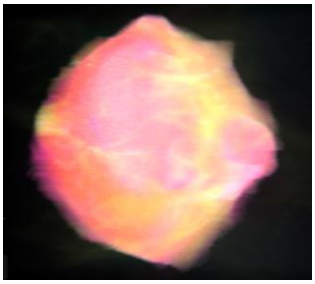
ECE 408 Project Report
3D Cell Volume Calculation
Jinghan Huang, Chao Xu, Run Zhang

1 Background

1.1 Problem Statement & Motivation

a) Problem Statement:

Nowadays, cell detection is much more improved and by electron microscope, we can obtain a point could data set for the 3d image of the cell. And with light or the help of chemical reagents, the shape of the cell can show off and so does other internal structures including DNA and etc. In the below screenshot, there is a clear outline for the cell with the membrane surrounding. By knowing the position of the cell membrane, we can find the cell and then need to calculate the cell volume. As a parallel coding project, finally we need to improve our code with parallel method to reduce running time.



3D Cell Image (tiff format)[1]

b) Problem Motivation:

How can we know the volume of an irregular object? Here is a reasonable method: we can put it into the water and then observe the volume change before and after we drop the object. Though it is easy to do and reliable, how can we detect the volume of an irregular cell? It is impossible to apply previous method to deal with tiny object like cells. So, we need to rely on the power of computer to scan the whole point cloud dataset, finding the membrane position and then calculate the internal cell volume (convert pixels to volume). And using parallel cuda code, we can achieve a fast and efficient way to calculate the cell volume.

1.2 Solution Process

a) Cell Decision:

The first step is that we need to locate the cell position and its shape. And cell membrane should be the outline of the cell. So, find the membrane positions surrounding the cell, which should be a close figure to indicate the shape of the cell. By figuring out the membrane shape and position, we can find whether the volume (pixels) is inside or outside the cell basing on the cell membrane.

b) Pixel Number:

Secondly, by knowing the cell membrane(outline), we can figure out pixels (or points) inside the cell. Count the pixel number within the cell membrane. With the

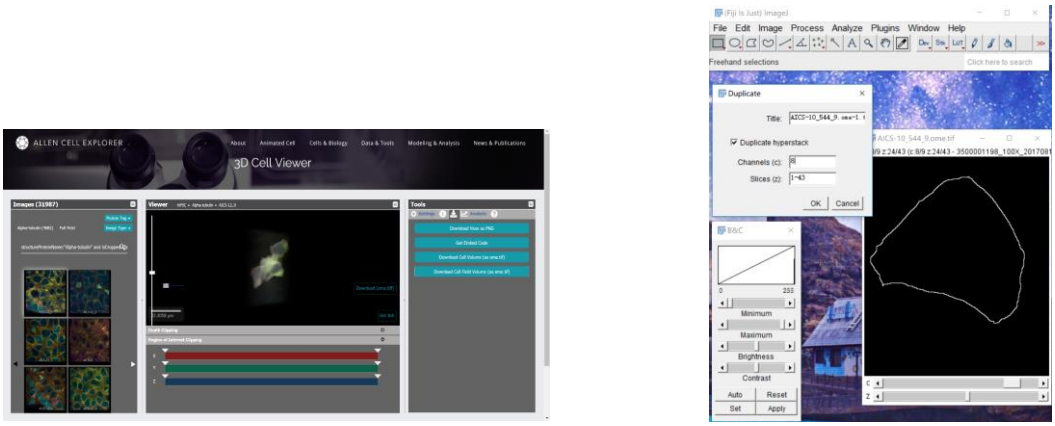
provided physical size for each pixel in the cloud point dataset, we can calculate the cell volume by multiplying the internal pixel number with the physical size of each pixel.

c) Parallel Code Design:

The most significant step is using parallel code to implement our program, in which way we can save the running time and improve the efficiency of cell volume calculation. And how to implement our serial code into parallel code and its effect will be introduced later.

1.3 Introduction to Allen 3D Explorer

Basically, our data source comes from the website named ALLEN CELL EXPLORER. Within the website, there is a 3D Cell Viewer shown in below screenshot. For those cell data tagged "Alpha-tubulin", cell membrane is marked and shown clearly in the image and its point cloud dataset is provided in TIFF format.



Left: Source from 3D Cell Viewer of Allen Cell Explorer[1]
Right: Membrane Segmentation Figure in X-Y Plane

And we can use the software ImageJ to view the basic information of provided dataset, including the physical size of a single point and view the cell image for each layer in X-Y plane. In addition, there are 9 channels provided for the cell dataset for different chemical reagents' view point. And Channel #8 show the cell membrane, the outline of the cell as shown in the below screenshot. In this way we are able to find the internal point number for each layer and sum up every layers' points to obtain the total volume.

1.4 Data Preparation

Though we can view the image and information of the cell, TIFF format is not a convenient access for C code to read the data. Therefore, we convert tiff Format into .csv and .png file for every layer (X-Y Plane as 2D array). For the usage of cell detection and cellular volume calculation, we improve the code to provide 2d arrays in PNGs and CSVs of each layers for the point cloud dataset. There is also a dir.csv to store the names of cell folders and each cell folder has a size.csv contains the size information of the point cloud data. And then in our cell detection and volume

calculation, we can use every point cloud dataset conveniently.

For the details, it takes a number of pages for us to discuss it. Please check "Point_Cloud_Dataset.pdf" for more information.

2 Sequential Method

2.1 Initial Method to Solve this Problem

For every cell, we can observe the image of this cell in the xy plane at different z values. For the brute-force method, we will make this cell into slices of a thin length through the z-axis. Then we will go through each pixel in the xy plane and add them together to get the area of the cell in this slice. Next, we will use the following formulas to obtain cellular volume. For a $h * w * t$ cube containing one cell,

$$\oiiint num(cell\ voxels) * V_{voxel} = \int_{z=0}^{z=t} dz * \oint\oint_{x=0,y=0}^{x=h,y=w} IsCellPixel(x,y) * S_{pixel}$$

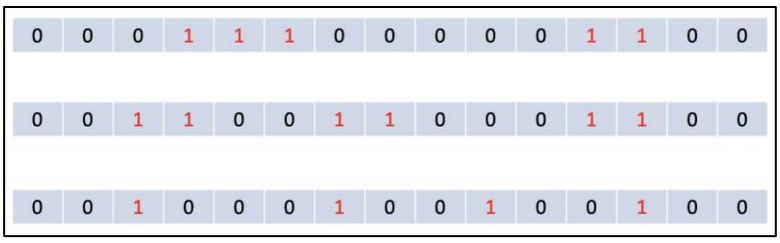
$$Volume_{slice-i} = Area_{slice-i} \times Thickness$$

$$Volume_{cell} = \sum_{i}^{n} Volume_{slice-i}$$

2.2 The Design of Sequential Code

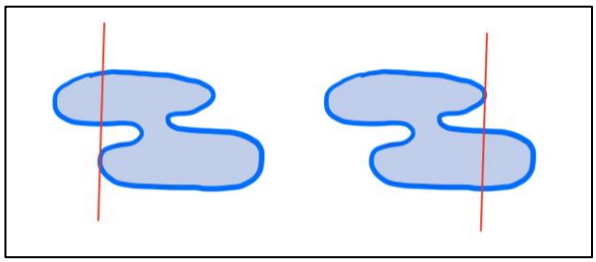
When we design our sequential method, we need have long-term consideration except for the functionality. It should be easy to be implemented by using parallel method afterwards. At first, we are motivated by parallel scan method. Because we need to add many numbers when we calculate the cellular volume, we can use Parallel Scan method. For Example, if we need to add four points (x_1, x_2, x_3, x_4), we let $x_2 = x_1 + x_2$ and $x_4 = x_3 + x_4$. Finally, $x_4 = x_2 + x_4$. The answer will be stored in x_4. In this case, we can calculate cellular volume in parallel method.

However, the actual situation is more difficult than we expect. We need to handle many special cases. Because the code read the image line by line, the information will be restrictive. The following figure can be regarded as the example to discuss about the limitations of this sequential code.



The data in the image in one line

In this figure, the red number "1" is used to represent the membrane pixels (white points) and the black number "0" is used to represent the pixels inside and outside the cell (black points). It is obvious that we will not count the number "0" at beginning and end of the line, because these pixels must be outside the cell.



The multiple possibilities for the line 2 in the above figure

However, for the numbers “0” in the middle part, it is hard to decide whether it should be counted or not, because there exist various possibilities.

2.3 Challenges

a) Innovation

The biggest difficulty is that there are no existing methods in the lecture for our reference. Even on the Internet, we can hardly find related methods. It really tests our innovation. We need to come up with our own ideas almost from scratch to solve this problem.

b) Decision

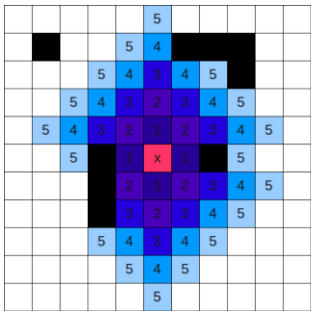
In our data, only membrane is represented as white. The pixels inside the cell and outside the cell are both represented as black. However, it is difficult for us to decide if the pixel we currently choose is inside the cell or outside the cell.

c) Communication

Since we need to decide whether the black points are inside the membrane or not, the relationship between the position of this point and all the white points is very important. Therefore, we need communication among these points. When we detect white points, we need to mark some black points, and the marked black points will be counted. As we know, global memory is the best choice for communication. However, it takes a very long time to access global memory. Therefore, how to minimize global memory with good communication is a big challenge for us.

3 Parallel Method

3.1 Motivation & First CUDA Trial



Example of flood fill starting from the seed in red and the “membrane” in black[2]

Parallel Method Motivation:

"A single spark can start a prairie fire." Chairman Mao's famous saying means that great military power could grow from small guerrillas. It is like a single spark could spread and become a prairie fire through the "flood fill." Basically, in the nature, such a process is parallelly executed in nature. So what about doing flood fill/BFS in parallel in cuda kernels.

Begin from Innovation:

a) Try BFS/Flood Fill from some fixed location (e.g. center) of the block to visit intra/extra-cellular space:

Basically, we calculate the volume by layers/levels and sum them up. For each layer/level, first we provide a point within the cell membrane and mark it with level 1 (numbers in the figure below should be decreased by 1). By BFS, we will visit its neighbors in four directions (up, down, left and right). As long as its neighbor pixel is neither visited/marked nor at the membrane border of the cell, we will mark the neighbor pixel as level 2, then level 3, level 4,… Until all points reach the boundary of the cell, the sum of the current pixels labeled with level would be the area of current layer (multiplied by the physical volume size of one pixel).

b) Pixels on the same level – working threads:

Within one block, each thread will be assigned to one point of the image and inactivated initially. During the BFS scanning, there is a "while" loop for BFS searching, and each step will increase the level by 1. Within each loop step, if the thread is marked with current level number, it will be activated and increase the block volume as long as it is not on the membrane. Therefore, the number of threads activated will increase as the level increases, which should be a faster way to do BFS than in serial code.

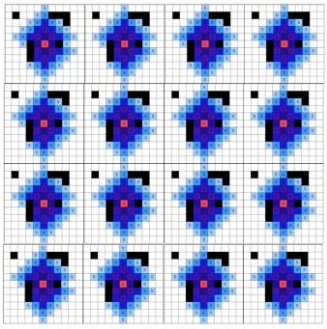
c) One block for one cell slice, thus no need for inter-block communication:

In our first code version, we only use one block for each layer and the total block size is no less than the pixel number of our given layer cell image. To be specific, in our Cuda kernel, currently we design the program with the block size of 1024 (32 pixels*32 pixels) and ceil(imageWidth/blockWidth)*ceil(imageHeight/blockWidth) blocks to deal with one layer of the membrane picture. It is always an issue that the given image size if larger than 1024, so that we need to resize the membrane image to satisfy the block size.

Limitation:

a) The usage is limited since there could be only 1024 threads per block at maximum. Since the maximum limit of our GPU is 48KB shared memory per block and 1024 threads per block, and our image size of about 500*500 pixels, we cannot run an image within one block or store shared memory for image size.

3.2 Second CUDA Trial



Flood Fill starting from the seed in multiple blocks[2]

Improved from initial trial:

a) Divide each layer ($X-Y$ plane) into multiple blocks:

To solve the previous limitation for image size, we apply multi-blocks design this time, thus dividing large images into small blocks and then assigning every thread to one pixel of the image. So, we can obtain non-membrane volume for each block by parallel code.

b) Run BFS from the center of blocks in parallel:

Using the same BFS as before, run BFS within each block by starting at the center of the block. Here we read the cell grayscale data and write the volume detected for each block directly.

c) Use the idea of disjoint set to establish up-trees for blocks inside and outside the membrane in parallel:

By now we only know whether a block includes membrane or not according to its block volume, but we do not know whether a block is inside the membrane or not. Therefore, we decide to connect all blocks outside the membrane. Up-tree is a convenient way to establish disjoint sets (one set outside the membrane while others inside the membrane). For each block, if its upper or left block is not on the membrane (full block volume calculated), it will choose its upper or left neighbor as parent node. Then update each block's parent block to the root parent and blocks whose parent node is at the left upper corner will be the blocks outside the membrane. By summing up all the block volume outside the membrane, we can get the complementary of the cell volume.

Limitations:

a) Useful for more cases but error exists in up-tree design:

For up-trees, each block would search its upper or left block for its parent block. However, exceptions occur as long as some blocks cannot reach the left-upper corner by only searching left or up.

b) Hard to know the block's volume along the membrane

In the method above, we ignore the blocks containing membrane pixels, which could have some partial volume that belongs to the outside volume.

c) Waste of time to sum up for external blocks:

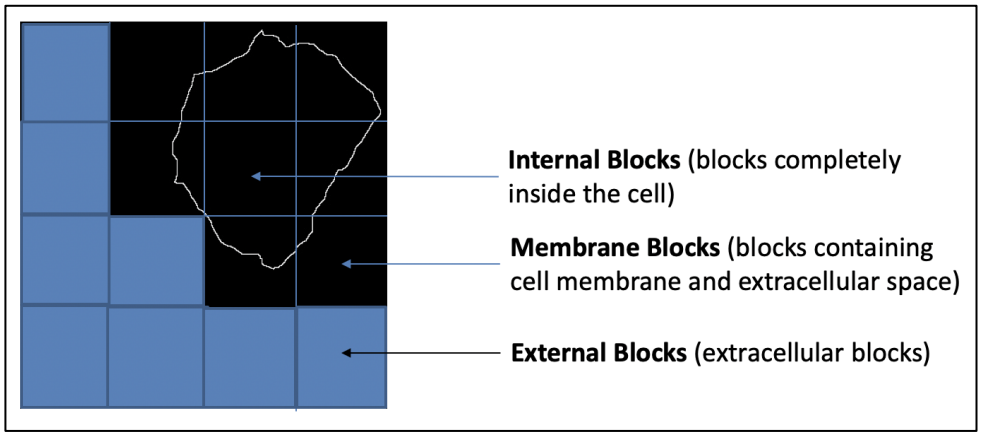
The basic idea of using BFS is to make the volume more accurate by knowing the tiny volume part along the membrane blocks. But if we only need to detect if the blocks are in the membrane or not, BFS is not necessary to calculate the block volumes.

3.3 Final Design Overview

Improved from previous design, we implement our program with four steps: firstly, identify whether the block is on the membrane; secondly, connect all the blank internal blocks and all external blocks respectively; thirdly, traverse blocks along the membrane to calculate the extracellular volume; finally, sum up the volume of each plane to get the result. For a $h * w$ plane containing one cell slice,

$$S_{cell} = h * w - BlockWidth^2 * Number\ of\ Internal\ Blocks$$
$$-Number\ of\ extracellular\ pixels\ in\ Membrane\ Blocks\ [Pixels]$$

For the simplicity of expression, we define three kinds of blocks as following:



Block Category – Defining Three Kinds of Blocks

a) CUDA Kernel 1 (Plain Sum): Identify membrane blocks:

Developing from the previous trial, we need to identify whether a block is a membrane block. This time we use each thread to detect if the pixel is 255 (grayscale value for membrane pixel). And as long as all the pixels in one block are 0 (grayscale value for non-membrane pixel), this block should not be on the membrane; otherwise, it is a membrane block. In this way, we can identify membrane blocks more efficiently and avoid wasting time on running BFS in non-membrane blocks, i.e. internal blocks and external blocks.

b) Sequential: Use Disjoint Set to connect all the external blocks (Distinguish external blocks from non-membrane blocks):

In the first cuda kernel, external blocks have the same feature as most of the internal blocks: all pixels in the block are blank (with grayscale 0). Disjoint set is employed to connect all those blank blocks together. Specifically, blank internal blocks are in one set, and all the external blocks are in the other set. Details and advantages of sequential disjoint set algorithm are introduced in 3.4.

c) CUDA Kernel 2 (*Optimized* Flood Fill): Flood fill/BFS traverse the extracellular

space in the membrane blocks

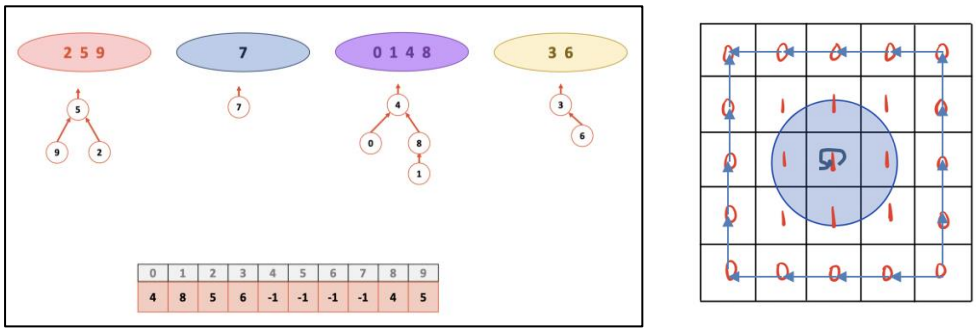
To obtain the extracellular volume, we also need to calculate the extracellular volume of the membrane block besides volume of extracellular blocks. Thus, some searching seeds are initialized in the membrane blocks to BFS traverse the extracellular space. See details in 3.4.

d) Final Sum:

Eventually, by identifying each block's position (inside or outside or on the membrane) and calculating those partial volumes in the membrane blocks and outside the cell, we can sum up all the block volumes outside the membrane and also those partial volumes outside the cell but in the membrane blocks. Then we get the total volume outside the cell, and the total volume inside the cell should be the image volumes minus by the calculated total volume outside the membrane.

3.4 Disjoint Set for Identifying External Blocks

In the field of computer science, disjoint set is usually employed to efficiently maintaining partitions of a set subject to. That is, it is specialized for *find* (identify the belonging set) and *union* (merge two sets) operations.



An Disjoint Set[3] Example (Left) the Disjoint Set in Our Sequential Code (Right)

· Mark the External Blocks as 0
· Mark other blocks (Internal + Membrane) as 1

Initially, each block is a single set. Each block will reach out and try to union with adjacent non-membrane blocks. Finally, all the external blocks will be in the same set as the figure above shows. We add padding blocks around the cell for optimization, which will be explained later. Besides, path compression is actually used in coding but not drawed in the figure for simplicity. The arrows show the parent-child relationship. Each membrane block is a single disjoint set; so, we do not draw arrows on them.

Why do not we use BFS/DFS in this sequential part? After all, it could be also handled as a linear-time traversal problem outside the cell. However, disjoint set has some noticeable advantages over BFS/DFS in this specific task:

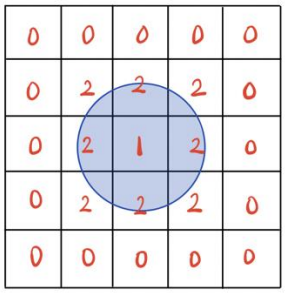
- Less space occupied
  Using disjoint set, only parent of each block should be recorded in space, which has the same size as a recording-visit array for each block in BFS/DFS. We do

not need to create an extra queue for BFS nor exploit system stack in DFS.

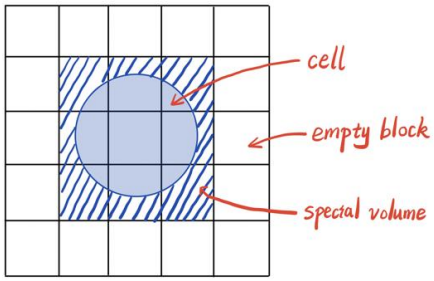
- Shorter run-time (even smaller constant in $O(n)$)
  Disjoint set (using path compression) could actually work in only time $O(n)$ in our task, which has the same time complexity as BFS/DFS. n refers to the number of blocks in one slice. When we scan blocks from upper to lower, left to right, each block will unite with the upper left block if the order of checking neighbors is specified as (left, upper, right, lower). Since the find operation could be compressed, i.e. building a 2-level tree for each partition, each block could find the ancestor in just one step. Thus, each union/ find operation is truly quick-union/find, costing $O(1)$ time. Furthermore, this constant is generally smaller than DFS/BFS.

3.5 Flood Fill the Membrane Blocks

Algorithm Route Overview



· Mark the Membrane Blocks as 2



· Run optimized BFS in the membrane blocks (marked 2) to calculate the special volume (volume outside the membrane that we miss before)
· Sum up two parts of volumes outside the membrane, which is the complement of the cell volume.

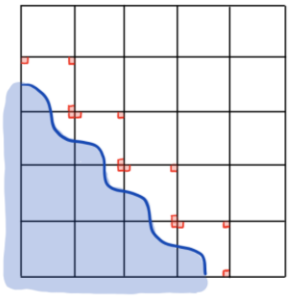
Optimized Flood Fill/BFS: a single spark could start a prairie fire. What about more?

While in the previous trails, there is considerable volume deviation caused by neglected membrane blocks; our final design can precisely calculate the extra-cellular space in membrane blocks in most cases with better parallelism.

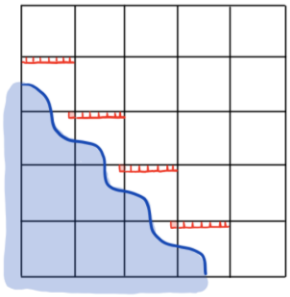
We design and compare three strategies to setup the BFS seed threads in the membrane blocks as shown in the figures below.

Seed Initialization Strategies:

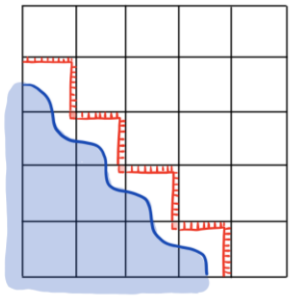
- Corner seeds: In the strategy #1, we only initialize seeds at four corners of each block. If one corner pixel is not a membrane pixel and adjacent (in eight directions) to an external block marked in the previous steps, this pixel/thread will be initialized as a BFS seed (marked red in figures below).
- Edge (upper and lower) seeds: In the strategy #2, we extend seed threads from corners to upper and lower edges. That is, if one pixel on the upper or lower edge is adjacent to an external block, it will be initialized as a BFS seed.
- Edge (full) seeds: In the strategy #3, we extend seed threads to all four edges.



#1. Corner Seeds



#2. Edge (Upper & Lower) Seeds



#3. Edge (Full) Seeds

Strategy Comparison:

	Function Name	Duration (µs)	Grid Dimensions
parallelBFS			
1	parallelBFS	463.335	(41, 84, 62)
2	parallelBFS	437.443	(54, 57, 63)
3	parallelBFS	328.098	(50, 75, 41)
4	parallelBFS	167.425	(37, 42, 50)
5	parallelBFS	139.714	(37, 31, 55)
plainSum			
6	plainSum	316.100	(41, 84, 62)
7	plainSum	281.986	(54, 57, 63)
8	plainSum	224.001	(50, 75, 41)
9	plainSum	115.040	(37, 42, 50)
10	plainSum	93.985	(37, 31, 55)

Strategy #1

	Function Name	Duration (µs)	Grid Dimensions
parallelBFS			
1	parallelBFS	439.112	(41, 84, 62)
2	parallelBFS	411.011	(54, 57, 63)
3	parallelBFS	309.314	(50, 75, 41)
4	parallelBFS	158.209	(37, 42, 50)
5	parallelBFS	131.201	(37, 31, 55)
plainSum			
6	plainSum	316.677	(41, 84, 62)
7	plainSum	282.978	(54, 57, 63)
8	plainSum	223.425	(50, 75, 41)
9	plainSum	114.081	(37, 42, 50)
10	plainSum	94.401	(37, 31, 55)

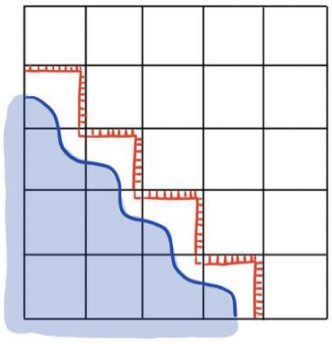
Strategy #2

	Function Name	Duration (µs)	Grid Dimensions
parallelBFS			
1	parallelBFS	425.735	(41, 84, 62)
2	parallelBFS	399.427	(54, 57, 63)
3	parallelBFS	301.282	(50, 75, 41)
4	parallelBFS	154.050	(37, 42, 50)
5	parallelBFS	127.745	(37, 31, 55)
plainSum			
6	plainSum	313.894	(41, 84, 62)
7	plainSum	281.442	(54, 57, 63)
8	plainSum	225.538	(50, 75, 41)
9	plainSum	115.073	(37, 42, 50)
10	plainSum	95.360	(37, 31, 55)

Strategy #3

Time Cost Comparison of Three Seed Initialization Strategies

We use five 3D cell data to test our three strategies. Generally, we get the result of consuming time: Strategy #3 < Strategy #2 < Strategy #1. If we use Strategy #2, it saves 5.67% time compared with Strategy #1. If we use Strategy #3, it saves 8.32% time compared with Strategy #1. We will analyze the difference of these three strategies specifically as follows.



Our Final Choice for BFS Seed Initialization Strategy

- Fewer BFS/Flood Fill searching levels

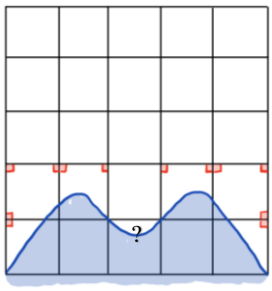
As shown in above figures, red points refer to the start points for BFS, i.e. more red points mean there are more start point in the first level of BFS. We tried different numbers of start points as above. And seen from the running time record, the running time decreases as the start points of BFS increases. As the start points increase, there would probably be less search level, so that it will help with improve the running time of BFS.

- Less control divergence

Control divergence varies for three situations and different block size. In Strategy #1&3, control divergence exists since each thread is activated by BFS level and is very possible to have multiple threads activated in different warps at the same time. So, with some warps having all the threads activated, many warps have some threads activated by BFS but others inactivated. For Strategy #2, we start BFS from the top side of the block to the bottom side. As the block size be 32, it is possible that there would be no control divergence at the beginning of BFS until we reach the membrane pixels. In conclusion, it is more likely to be less control divergence in Strategy #2 but more in Strategy #1&3.

- Higher precision

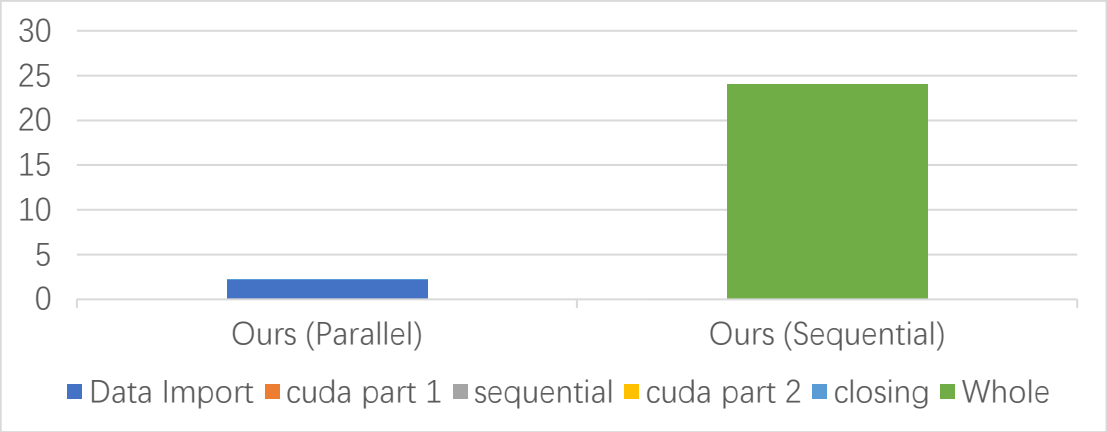
For strategy #1, only seeds at four corners will be initialized. Some special cases cannot work, for example, like the picture shown below, where the middle-bottom block has its corners occupied by membrane, so that BFS cannot work in that block (stopped at the beginning by membrane at the corners). However, in the Strategy #2 & #3, with more start points for BFS, it is less likely that all the start points would meet the membrane and stop by exception. Therefore, with more starting points, the precision of final results on volume calculation would increase accordingly.



The Killer Case for Strategy #1

4 Parallel Optimization

By comparing the time between sequential and parallel code, the parallel version is much more improved from the sequential version.



Sequential vs. Parallel Running Time Comparison

4.1 Memory Privatization

Coding Rule of Thumb: Each global memory location is at most accessed (read + write) once by each block.

In our previous trials, we want to use one single cell to run the whole image, whose size is about hundreds by hundreds of pixels. However, the maximum shared memory size per block is 48KB, which is relatively small and not large enough to contain data in image size.

```
int prev_vol = -1;

for (uint level = 1; prev_vol != global_vol[z]; level++) {
  prev_vol = global_vol[z];
  // if the thread is on the level of BFS, it starts to search
  if (dist[tx][ty] == (uint)level) {
    for (int dir_idx = 0; dir_idx < 4; dir_idx++) {
      // neighbor pixel's index
      next_tx = tx + dir[dir_idx][0];  next_ty = ty + dir[dir_idx][1];
      next_x = x + dir[dir_idx][0];    next_y = y + dir[dir_idx][1];
      // if the neighbor pixels are not visited before
      if ((next_tx >= 0) && (next_ty >= 0) && \
        (next_tx < blockDim.x) && (next_ty < blockDim.y) && \
        (cell[next_tx][next_ty] == 0) && \
        !atomicCAS((uint *)&dist[next_tx][next_ty], (uint)0, (uint)level + 1))
          atomicAdd(&global_vol[z], 1);
      }
    }
  }
  __syncthreads();
}
```

Codes only using Global Memory

To deal with large images, we improve our code to use multiple blocks as mentioned above, and in that way, we can use shared memory for each block to reduce global reading and global writing times, thus decreasing running time for parallel code. Besides shared memory, we also use local variable to help us monitor the volume change for the current thread. The number of atomicAdd operations is also decreased.

```
__shared__ uint shared_vol = 0;
int prev_vol = -1;
__syncthreads();
uint local_vol;

for (uint level = 1; prev_vol != shared_vol; level++) {
  local_vol = 0;
  prev_vol = shared_vol;
  // if the thread is on the level of BFS, it starts to search
  if (dist[tx][ty] == (uint)level) {
    for (int dir_idx = 0; dir_idx < 4; dir_idx++) {
      // neighbor pixel's index
      next_tx = tx + dir[dir_idx][0];  next_ty = ty + dir[dir_idx][1];
      next_x = x + dir[dir_idx][0];    next_y = y + dir[dir_idx][1];
      // if the neighbor pixels are not visited before
      if ((next_tx >= 0) && (next_ty >= 0) && \
        (next_tx < blockDim.x) && (next_ty < blockDim.y) && \
        (cell[next_tx][next_ty] == 0) && \
        !atomicCAS((uint *)&dist[next_tx][next_ty], (uint)0, (uint)level + 1))
          local_vol ++;
    }
    atomicAdd(&shared_vol, local_vol);
  }
  __syncthreads();
}

if (tx == 0 && ty == 0)
  atomicAdd(&global_vol[z], shared_vol);
```

Code using Local+Shared Memory

If we try to apply memory privatization, it saves 11.03% time compared with using global memory.

	Function Name	Duration (µs)	Grid Dimensions
parallelBFS			
1	parallelBFS	545.129	{41, 84, 62}
2	parallelBFS	423.331	{54, 57, 63}
3	parallelBFS	320.482	{50, 75, 41}
4	parallelBFS	162.817	{37, 42, 50}
5	parallelBFS	131.169	{37, 31, 55}

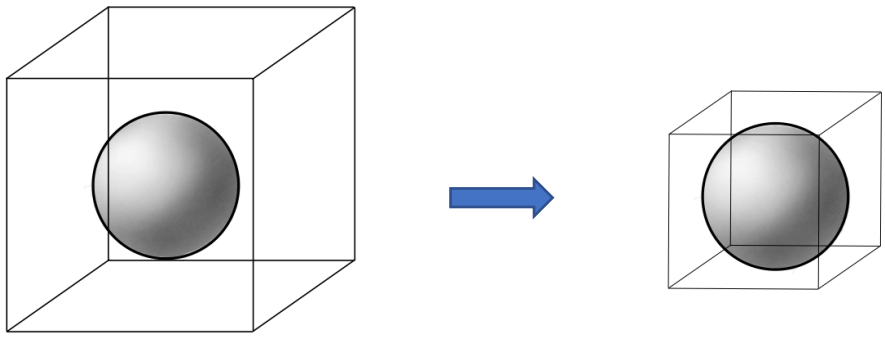
Global Memory

	Function Name	Duration (µs)	Grid Dimensions
parallelBFS			
1	parallelBFS	425.735	{41, 84, 62}
2	parallelBFS	399.427	{54, 57, 63}
3	parallelBFS	301.282	{50, 75, 41}
4	parallelBFS	154.050	{37, 42, 50}
5	parallelBFS	127.745	{37, 31, 55}

Shared Memory

The Comparison of Time for using Global Memory and using Shared Memory

4.2 Crop the Data of Cell to Smaller Size



The Crop of the Cell

For the given point cloud dataset of the cell, the cell would probably occupy only part of the image. But in previous code, we need to do memory read/write between device and host for the whole image. To decrease that part of time, we try to decrease image data size for each cell as small as possible, i.e. we can cut another cuboid inside

the image which also contains the whole cell. To be more specific, since each layer is separate, we shift every layer's membrane part to the left-top corner of the cut image. In this way, we can decrease the import cell size for our program, so that we can save much time in data memory copy from host to device or from device to host when we start a cuda code.

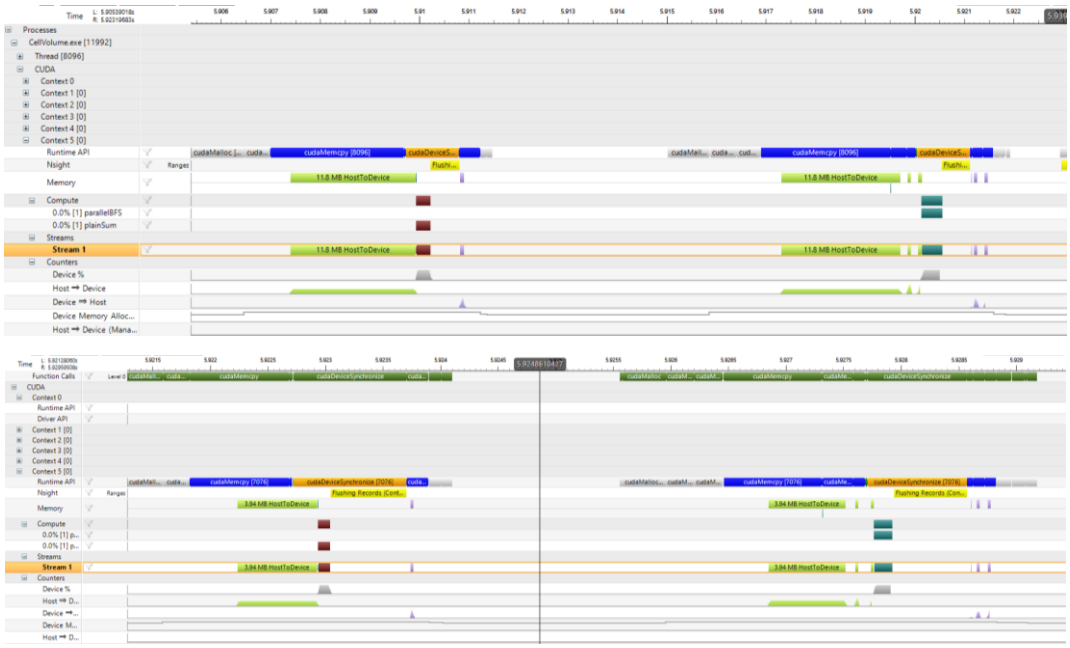
Two GPU information charts below show the differences in the running time of cuda memory copy, i.e. cudaMemcpy. It shows that after cropping the cell, there is huge running time improvement.

	Source	Destination	Duration (µs)
1	Host Unpinned	Device	2,738.078
2	Host Unpinned	Device	2,615.560
3	Host Unpinned	Device	2,520.816
4	Host Unpinned	Device	2,392.911
5	Host Unpinned	Device	1,957.613
6	Host Unpinned	Device	1,885.707
7	Host Unpinned	Device	868.006
8	Host Unpinned	Device	852.231
9	Host Unpinned	Device	654.084
10	Host Unpinned	Device	646.244
11	Host Unpinned	Device	70.658
12	Host Unpinned	Device	70.401
13	Device	Host Unpinned	65.825
14	Device	Host Unpinned	65.281
15	Device	Host Unpinned	64.833
16	Host Unpinned	Device	64.257
17	Host Unpinned	Device	64.257
18	Device	Host Unpinned	59.328
19	Device	Host Unpinned	59.297
20	Device	Host Unpinned	58.560
21	Host Unpinned	Device	51.329
22	Host Unpinned	Device	51.264
23	Device	Host Unpinned	47.425

	Source	Destination	Duration (µs)
1	Host Unpinned	Device	691.780
2	Host Unpinned	Device	665.956
3	Host Unpinned	Device	350.182
4	Host Unpinned	Device	282.981
5	Host Unpinned	Device	131.552
6	Host Unpinned	Device	131.329
7	Host Unpinned	Device	63.265
8	Host Unpinned	Device	63.040
9	Host Unpinned	Device	57.569
10	Host Unpinned	Device	57.376
11	Host Unpinned	Device	22.624
12	Host Unpinned	Device	22.497
13	Device	Host Unpinned	20.481
14	Device	Host Unpinned	20.160
15	Device	Host Unpinned	19.680
16	Host Unpinned	Device	14.144
17	Host Unpinned	Device	14.144
18	Device	Host Unpinned	13.280
19	Device	Host Unpinned	12.608
20	Device	Host Unpinned	12.096
21	Host Unpinned	Device	9.408
22	Host Unpinned	Device	8.864
23	Device	Host Unpinned	8.000

Time Cost Comparison of cudaMemcpy Pre-Crop (Left) and Post-Crop (Right)

In addition, it helps with saving the memory space in cuda from 11.8MB to 3.94MB as shown in the figures below (tagged stream 1).



Timeline Comparison Pre-Crop (Top) and Post-Crop (Down)

4.3 Extend the Image X&Y Size to the Multiple of Block Size and Add Paddings to the Image

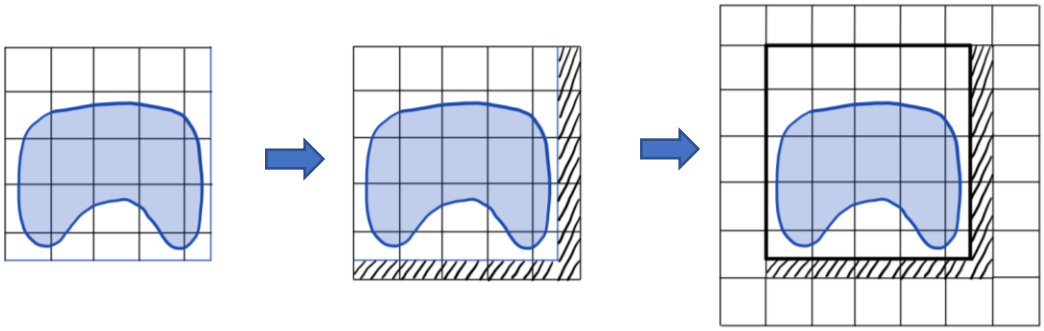
After cutting the cell image, there are two steps to extend image size:

a) Decrease control divergence by extend image size to the multiple of block size:

The random image size would probably cause control divergence at the right and bottom side of image. By extending image size to the multiple of block size, we can get rid of "if" which is used to check the boundary of the image to avoid control divergence for some warps. In this way we can improve our running time.

b) Avoid external blocks surrounded by membrane and the boundary of the image by adding paddings:

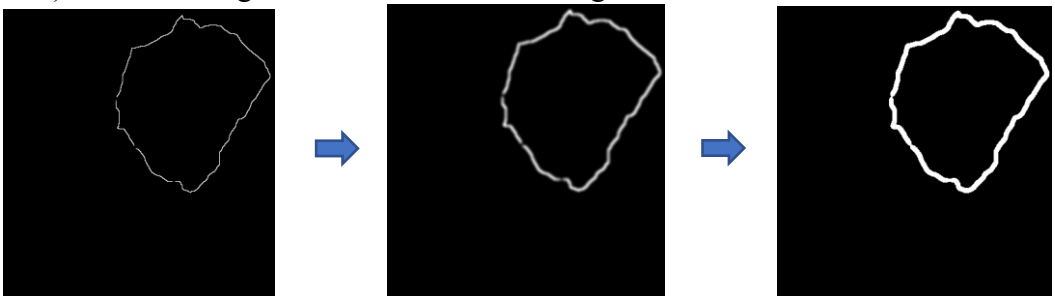
Though we has improved up-trees to avoid missing external blocks, if there are some external blocks surrounded by membrane and the boundary of the image, those block would not be able to connect with other external blocks, and therefore we might miss some volume when counting external volume. By adding external padding around the original image, we can guarantee that all external blocks could be connected together by avoiding exceptions shown below.



The Process of Adding Paddings and Extending blocks

4.4 Repair Broken Cell Membrane

Here is an answer provided to reply professor's question on how to deal with cell image with broken membrane. We apply Gaussian Blur in our cell image with some broken parts along membrane, which would probably cause errors in identifying internal and external pixels. By Gaussian Blur, the small broken parts along the membrane would be cover by the blur from neighbor pixels. Then by binarizing the blur image, we can recreate a cell image with complete membrane (a little thicker than before). The effect figures look like the following:



The Process of Repairing Broken Cell Membrane

5 Conclusion

Our creative idea for cell volume calculation in cuda parallel is trying to implement BFS in parallel code. Compared with sequential BFS, we can assign each point with one pixel so that the running time is much more improved. And in order to calculate the volume of an irregular cell, BFS is an efficient method to obtain accurate result. And we solve many problems in the process of converting sequential BFS into parallel version, including multiple blocks connection, membrane boundary volume detection and etc. In addition, to improve our parallel performance time, we crop the cell into smaller image size to decrease memory copy time. Moreover, we try multiple start points for BFS to achieve higher accuracy for the volume along membrane. Finally, we get really good result by achieving those optimizations.

We think this project tests our ability of innovation. Not only do we improve related skills, we also feel more interested in such kind of innovative projects.

References

[1] Allen Cell Explorer: https://www.allencell.org/

[2] Jung C-Y, Yoo S-L. Optimal Rescue Ship Locations Using Image Processing and Clustering. Symmetry. 2019; 11(1):32.

[3] CS225 Lecture Slides: https://wiki.illinois.edu/wiki/display/ZC2DS/Lectures