CPE810 Final Project: Depth Map From Multi-View Using Plane Sweep Algorithm

Haixu Song
ECE Department
Major in Applied Artificial Intelligence
10446032
hsong13@stevens.edu

***Abstract*—Multi-view modeling is widely used nowadays in several areas like auto-driving cars. As a CUDA Programming Course Project, I wrote the CUDA version of Plane-Sweep Algorithm, which is a classic algorithm used for generating depth map or points cloud. Firstly, I used 307 * 205 resolution pictures and python sequential code for testing and tuning. After finding the best parameters and source images, I wrote CUDA code and tested the kernel running time. Finally, we had a kernel running time of 55 milliseconds while processing five 3072 * 2048 images as inputs and outputting depth map in the same resolution.**

***Index Terms*—Multi-View Modeling, Plane Sweep Algorithm, CUDA, GPU RTX 2060, Python**

I. Introduction

A. Video-Based Rendering Background

Video-Based Rendering (VBR) is a classic research field that proposes methods to compute new views of a dynamic scene from video streams. VBR techniques are divided into two families. The first one, called off-line methods, focuses on the visual quality rather than on the computation time. These methods usually use a large amount of cameras and sophisticated algorithms that prevent them from live rendering. Therefore, the video streams are recorded to be computed off-line. The rendering step can begin only when the scene information has been extracted from the videos. Such three-step approaches (record - compute - render) are called off-line since the delay between acquisition and rendering is long in regard to the final video length. The methods from the second family are called on-line methods. They are fast enough to extract information from the input videos, create and display a new view several times per second. The rendering is then not only real-time but also live. Plane sweep algorithm is designed as an on-line method for video rendering. Because the algorithm is very suited for GPU and performs very high frames per second while using GPU.

B. 3D Modeling Background

3D modeling means you wanna generate 3D points cloud or depth-map from an angle of the real-world object. There are several ways doing this. It's widely used in several industries like movie and game industry, auto-driving industry, augmented reality industry, and even some scientific experiments. For example in movie industry. If a director wanna film a scene that the Empire State Building was destroyed, they will first generate a virtual 3D model of that building using 3D modeling techniques. Same thing happens in game industry. In auto-driving industry, if we want the algorithm to know how to make driving decisions, you have to feed the algorithm with 3D model or depth map of the front cameras. And this is required in a very high frame rate, since the car may drive through 20 meters or more in 1 second.

I'll just introduce some of those popular techniques nowadays.

1) LiDAR: Lidar means 'Light Detection and Ranging', which is a remote sensing method used to examine the surface of the object. Its working principle is mainly expressed as follows: a modulated laser signal is sent out through a laser transmitter, and the modulated light is reflected by the measured object and then received by the laser detector. The distance to the target can be calculated by measuring the phase difference between the emitted laser and the received laser.

LiDAR are widely used in auto-driving industry several years before when AI is not popular. The use of lidar in self-driving cars began with Sebastian Thrun. This person won the championship in the DARPA self-driving challenge more than ten years ago. Its early establishment of advantages started with lidar. Specifically, it is based on mechanical scanning lidar, and the manufacturer is velodyne. Later, Sebastian Thrun's team was acquired by Google and developed Google's self-driving car. The idea is also based on mechanical scanning lidar. The reason why Sebastian Thrun used lidar in that era is relatively easy to understand. After all, deep learning was not mature more than a decade ago, resulting in a limited improvement in effect of computer vision based on human feature engineering. In terms of computing power, CUDA has not yet been released and does not have enough computing power. The image output by the lidar at that time was directly a depth map, which could be used for obstacle avoidance and route planning without complicated calculations. Therefore, most of the self-driving cars made by several teams are based on lidar.

However, lidar also has some inherent shortcomings, which makes it not a very suitable solution for autonomous driving (the following is based on the data of velodyne 64-line lidar):

- Detection distance: The current effective detection distance of Velodyne's 64-line lidar is 120 meters and 360 degrees. When the on-board computer is fast enough,

the reserved brake reflection delay is still too small; relatively speaking, human eyes It's easy to spot threats in a driveway several hundred meters away.

- Detection accuracy: The working method of lidar is also discrete. Remember that the vertical and horizontal resolutions at a distance of 120 meters are already 0.3-0.5 meters, which means that a person standing at this distance may be ignored by lidar Yes, it can only be found by scanning at a closer distance.
- Scanning frequency: The highest scanning frequency of Velodyne 64 can only reach 15Hz, which is equivalent to 15fps of the video frame rate, and because it is 360 scanning, plus the time of data buffering and return, this delay is very scary, and it is impossible to make automatic Drive a car to increase the speed. At a speed of 120 kilometers per hour, a single scan of one frame can drive the vehicle out to 2.222 meters. In the same period, the industrial camera used in the car usually starts at 120Hz, and the camera with 1000Hz is also used. Not difficult to buy.
- Lifespan: It is an optical device that is constantly rotating all day long. It is not difficult to imagine the lifespan.
- Perverted price: The Velodyne 64, the ex-factory price was 70,000 US dollars, and it was about 100,000 US dollars in China.

2) Stereo Cameras: Many companies that develop autonomous driving technology use the three major components: Lidar, millimeter wave radar, and differential GPS. The total price of these three prices is over 0.1 million dollars. The problem is that it is simply unrealistic to make self-driving cars based on this price. Millimeter-wave radars have fallen rapidly in the past few years, and mass-produced goods for automatic cruise and collision avoidance can be reduced to less than 1,000 dollars. In principle, differential GPS can be used with ordinary GPS. But the price of lidar is really unbeatable, and it has become an important price threshold for autonomous vehicles.

Stereo cameras means taking several pictures using different camera angles. The most popular used one is two camera depth estimation. It used two cameras facing parallel direction with fixed difference between it. The algorithm is quite simple. For an element on the reference image, we can estimate that pixel on different depth. For each estimation, we find the projection of that pixel on the matching image. Then we find all the projected pixel on the matching image and find which pixel best fits the original pixel on the reference image.

There are also other ways to estimate depth from multi-view images. Some guys use two-view cameras along with artificial intelligence to estimate depth. From the figure, you can see that this guy used OpenCV to recognize the same object on two cameras, then use the two-view camera principle to estimate the depth of the detected object.

3) Single Picture Based: Along with the thrive of the artificial intelligence these days, more and more new techs are developed to do this task. More people focus on generating depth using single pictures. I'll just simply introduce some of them.

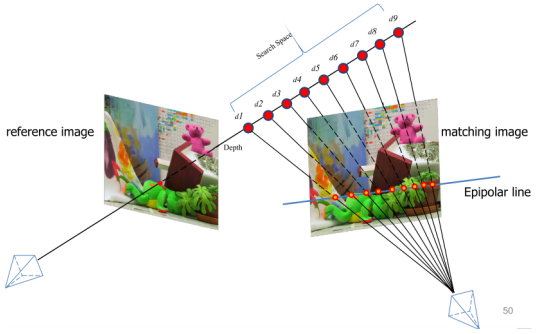

Fig. 1. Principle of Two View Depth

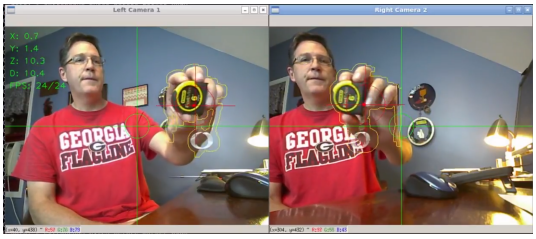

Fig. 2. Two View Camera With OpenCV AI

Video Based:

This is quite popular and widely used nowadays in all kinds of picture rendering and video rendering. They trained a RNN model and use them to estimate depth of current frame using the previous frames and current frame. You can see from the figure, they actually got quite a good result at last.

Fig. 3. Video Based AI Depth Estimation

Focus Based:

When you take a photo using a camera, you have to tune the focus length to get the most clear picture. However, when you tune the focus length, the object may gets clearer, other objects may get blurred. The focus based principle is using this property to estimate depth using blurring comparing within different focus length.

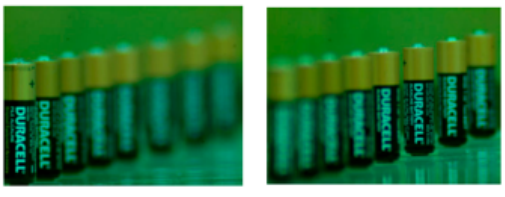

Fig. 4. Focus Based Depth Estimation

Pure AI based:

Some scientists also tried using pure single picture to estimate depth of it. They build a CNN and train the model using large data set. However, this process may not be explainable. Theoretically, one can not tell the depth using one single picture, so no matter how you train the model, the accuracy or scalability will be limited.

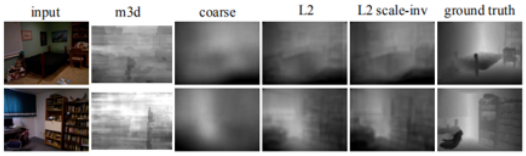

Fig. 5. Pure AI Based Single Picture Depth Estimation

C. Plane Sweep Algorithm

Let me introduce the plane sweep algorithm and its related optimized algorithms.

The original algorithm is used to generate depth or 3D points clouds using multiple aspect images of the same object. The principle is quite like the 2-view cameras but more complex. The plane-sweep algorithm provides new views of a scene from a set of calibrated images. Considering a scene where objects are exclusively diffuse, the user should "place" the virtual camera cam x around the real video cameras and define a near plane and a far plane such that every object of the scene lies between these two planes. Then, the space between near and far planes is divided by parallel planes Di as depicted in the figure. Consider a visible object of the scene lying on one of these planes Di at a point p. This point will be seen by every input camera with the same color (i.e. the object color). Consider now another point p' lying on a plane but not on the surface of a visible object. This point will probably not be seen by the input cameras with the same color. Figure 1 illustrates these two configurations. Therefore, the plane-sweep algorithm is based on the following assumption: a point lying on a plane Di whose projection on every input camera provides a similar color potentially corresponds to the surface of an object.

During the new view creation process, every plane Di is computed in a back to front order. Each pixel p of a plane Di is projected onto the input images. Then, a score and a representative color are computed according to the matching of the colors found. A good score corresponds to similar colors. This process is illustrated on the figure 7.

Then, the computed scores and colors are projected on the virtual camera cam x. The virtual view is hence updated in a z-buffer style: the color and score (depth in a z-buffer) of pixel of this virtual image is updated only if the projected point p provides a better score than the current score. This process is depicted on the figure 8. Then the next plane Di+1 is computed. The final image is obtained when every plane is computed.

So the algorithm will be coded like this: firstly we iterate each point on each plane, then project that point on all the input images. Then we calculate the loss or similarity score

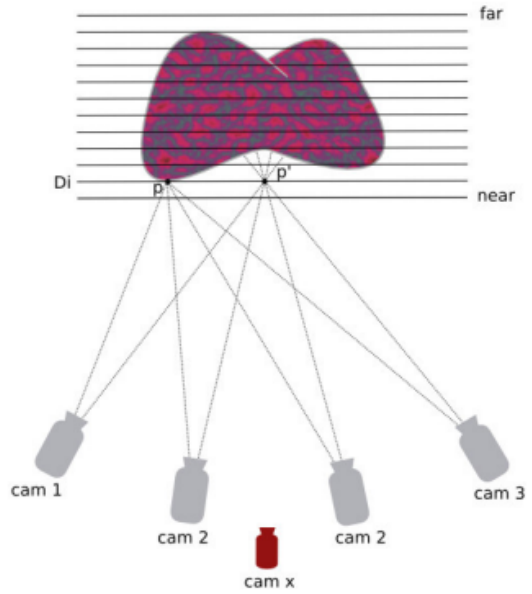

Fig. 6. Plane Sweep Algorithm Geometric Configuration

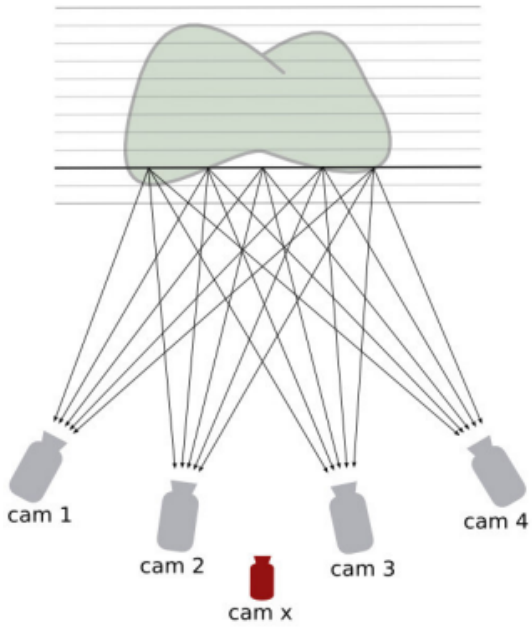

Fig. 7. Every point of the current plane is projected on the input images. A score and a color are computed for these points according to the matching of the colors found.

using the pixels projected. Finally, we find which plane has the greatest similarities within those projected pixels.

So what is the similarity function? The similarity function is used to judge how its alike when given a bunch of pixels. There's a lot of similarity functions. Some of them are based on pixel colors, while some of them are based on pixel window correlations.

Let's first talk about pixel color similarity. This is just a way to estimate how the pixels are alike in color. There are

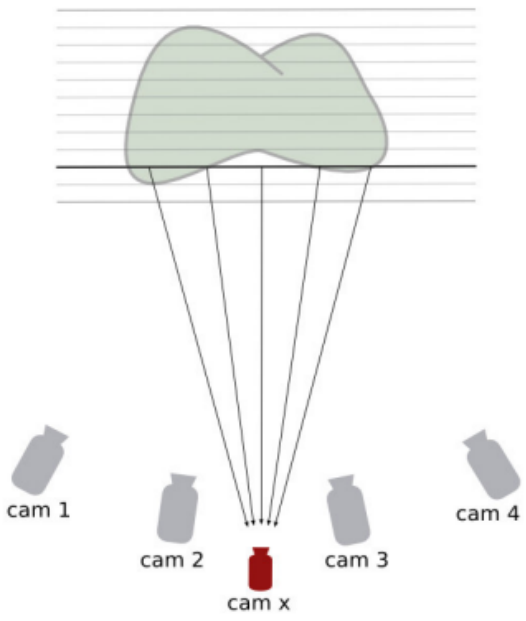

Fig. 8. The computed scores and colors are projected on the virtual camera.

several widely used functions like SSD, which means sum of squared difference, and SAD, which means sum of absolute difference. SSD and SAD both firstly calculate the average of the given pixels in all RGB channels, then accumulate squared or absolute difference together.

Pixel window correlation means how two given pixel window are alike in the color changing trend. In this way, we can estimate how the pixel windows are alike even if they have different theme colors. The commonly used correlation is cross correlation, which means you can only calculate correlation in pairs.

II. Sequential Coding

I used python to do the sequential version of plane sweep algorithm. The reason I used Python is that: I need to test if my equations works. I also need to tune those parameters like how many planes, planes depth range, window size, etc to find the best parameter to get the best result. Python is a script language which is very simple in syntax and have a lot of useful dependencies and libraries to help you make your algorithm work in short coding time.

So I wrote a Python model named "psalgo" which means plane-sweep-algorithm. In this model, I exposed two kinds of classes. One is called InputImage, and one is called TargetImage. I designed the API quite simple. Firstly, you should firstly initialize the all the InputImage you need with the source image path and input image's camera paramter. Then you should initialize the TargetImage with the target's camera parameter. You can optionally provide the ground truth points cloud file path. Then you have to load those InputImage instances into the TargetImage instance you just initialized. Finally, you fit the TargetImage instance with with the fitting parameters, like the window size, depth range and plane numbers.

A. Projection

So let's firstly talk about how to do the projection and the inverse projection. There are several important camera parameters. The first one is external parameter. It is a 3 times 3 square matrix. Each row and each column is a unit vector. External parameters represents the camera's direction of each axis. The usage of this matrix is that you can put a world coordinate on the right of it and you can get the coordinate relative to the camera's new axis. The matrix multiplication is just like doing the axis rotation since the external parameter matrix is always a rotation matrix.

The second important parameter is the camera's internal parameters. This is also a 3 x 3 square matrix. This matrix's top-left 2 x 2 is a diagonal matrix and the bottom-right of the matrix is always 1. The last number in line 1 and line 2 are always approximately half of the image resolution. The internal parameter matrix is always used to change the camera coordinate into the projected pixel position. When you times a 3 x 1 world coordinate on the right of it, you will get a 3 x 1 vector which represents the homogeneous coordinate of the pixel position. The third number of that result actually means the depth of the pixel. From the matrix multiplication, we can see that what the internal parameters did is just re-scaled the x and y of the camera coordinate using fx and fy, then did a shifting using cs and cy. The reason why shifting is needed is that the result after re-scaling is the relative position to the middle of the picture. We have to change the coordinate into commonly used position relative to the top-left of the pixel. So that's why the cx and cy are usually about half of the picture resolution.

$$\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

Fig. 9. Internal Parameter

The third important parameter is the camera position. It's simply just the camera's coordinate in the 3D real world.

So the projection equation is quite simple to understand now. If we call a 3 x 1 vector the world coordinate wld and a 2 x 1 vector the pixel coordinate pix. The process we find the pix using wld is called projection, and the process we find wld using pix is called inverse projection. So the projection process will be like this.

$pix = cart(ip * ep * (wld - cp))$

The equation is quite easy to understand. ip represents internal paramers, ep repersents external parameters and cp represents camera position. Firstly we find the relative position of object's world coordinate from camera. Then we do the

coordinate rotation using the external parameter to find the position of that object in camera's coordinate. Finally, we use internal parameter to get the actual pixel's homogeneous position on the image. The Cartesian coordinate of the homogeneous vector is a 2 x 1 shaped pixel position, whose first element represents the row and the second element represents the column.

So here comes the inverse projection process. We can simply put ip, ep and cp on the left and get the wld from pix. The equation will be just like this.

$wld = ep^-1 * ip^-1 * pix + cp$

B. Data set

The data I used is called fountain-P11. It's a very famous set of images which are widely used as benchmarks in MVS (multi-view stereo).

C. Sequential Code Result

I tuned 3 parameters in the sequential code, plane numbers, plane depth range and projection window size. Before started, I first changed the resolution of the source image into 0.01 of the original one. That's because the original image takes too much time to calculate the final result. Even though I reduced the resolution significantly, the running time still takes about 10 images for each depth picture.

This is the final conclusion I got.

- SAD similarity function and SSD similarity function has very similar output result.
  Theoretically, SSD do more penalty comparing to SAD, which mean it will be more sensitive on the less alike pixels. Since we choose the depth which has the most similarity and dismiss those with less similarity, how large the penalties are seems doesn't effect much when all the object surfaces are within range.
- With NCC (normalized cross correlation) similarity function, it performs better on edges comparing to SSD and SAD, but has more noise when the actual object is parallel to the plane.
  Cross correlation is high when two pixel windows are both located on the edge of the object. So the edges are detected better than SAD and SSD simply because NCC is born for edges.
- The larger the window size is, the smoother the depth image we got.
  If we calculate the similarity score using pixel windows, then nearby by pixel windows' scores are actually similar since most the pixels in their pixel windows are the same from the source image. With similar depth of near by pixels, we will get a smooth change depth image.
- The effective planes depths are within range 5 to 9.
- 60 planes are accurate enough for this depth estimation range. More planes looks the same.

III. CUDA Coding

The parallel version of this algorithm is like this. Each thread in GPU device process a single pixel point on the target

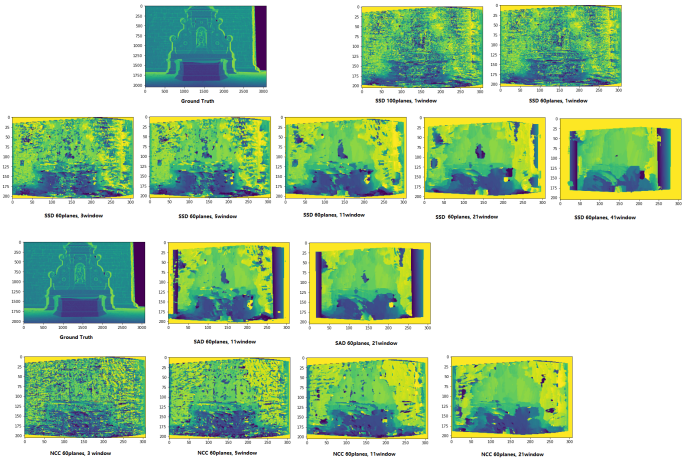

Fig. 10. Sequence Code Tuning

image. On each thread, we should firstly iterate each plane. In each iteration, we do the inverse projection from the pixel to the real world coordinate on that plane. After finding the real world coordinate, we should do the projection of that point on all source input images. Then we calculate the similarity score of those pixels we got from the projection. If the new similarity score of the current plane is larger than the previous minimum one, we update the minimum score using the current score and the depth estimation using the current plane depth. The whole process are shown in the figure.

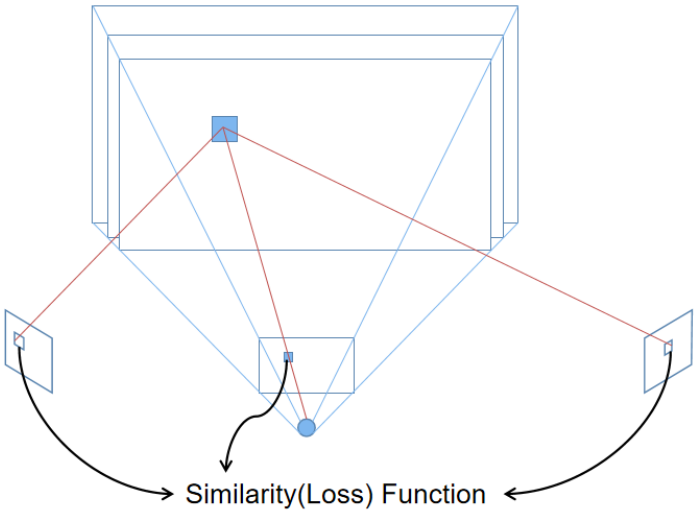

Fig. 11. Principle of Parallel Plane Sweep Algorithm

In order to make the CUDA kernel runs faster, we should optimize the algorithm and make each thread do the least float point operations. So the original algorithm's equation needs 2 matrix multiplication and 1 matrix addition or reduction when doing projection or inverse projection on each plane. So I changed the equation and reduced the matrix multiplication needed for the each thread. The changing is shown in the

figure.

$$pix = ip * ep * (wld - cp) \quad \rightarrow \quad P = ip * [\, ep \mid -ep*cp]$$
$$pix = Cartesian(P * homo(wld))$$

$$wld = ep^{-1} * ip^{-1} * pix + cp \quad \rightarrow \quad Pi = [ep^{-1} * ip^{-1} \mid cp]$$
$$wld = Pi * homo(homo(pix) * depth)$$

Fig. 12. The Equation Change

In the equation changing, I used many transferring between homogeneous coordinate and Cartesian coordinates. After the changing, each projection or the inverse projection only needs one matrix multiplication. Before changing, the whole kernel will need 5 * (2 * 3 * 3 * 3 - 3 * 3 + 3 + 2) = 250 float points operation on phase 1 (doing projection and inverse projection). After changing, the whole kernel will need 3 * 7 + 2 = 23 float points operation. So in all 60 planes, we reduced the projection float point calculation from 250 * 60 = 15000 into 23 * 60 = 1380. And that's really a huge reduction.

The similarity function I used is SAD. During the similarity calculation, I firstly iterated all the 5 pixels projected and find the average of them. Then iterate them again to get the sum of absolute difference. So the total float point operations are 5 + 5 * 3 = 20. So the total calculation within the whole kernel will be 1380 + 20 = 1400 for each thread.

So the final result we got is shown in the figure. You can see that the result is quite good not accurate enough. You can also see the textures on the surface. However, the surface of those background brick wall is not as good as expected. Since we used 60 planes ranged from 5 to 9, the resolution will be just (9 - 5) / 60 = 0.067m = 6.7cm. The brick texture is sure changing smaller than this range.

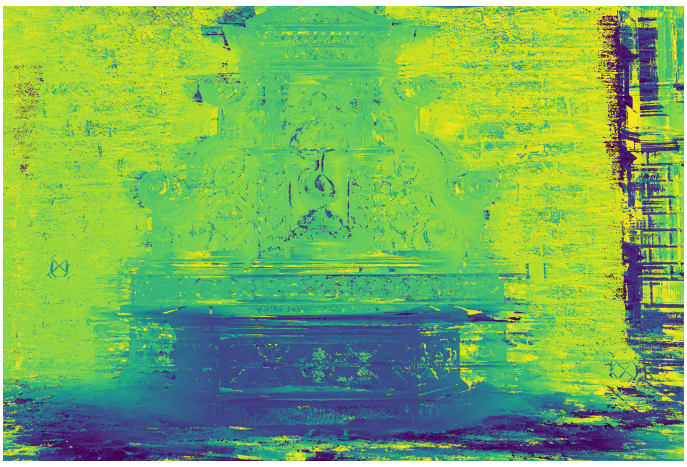

Fig. 13. The CUDA Kernel Result

The running time of the kernel function is just 56 milliseconds on my CUDA device, an NVIDIA RTX 2060 graphic card. The total float point calculation of this kernel is 3072 * 2048 * 1400 = 8,808,038,400. So the overall FLOPS is 8,808,038,400 / 0.056 = 157,286,400,000 = 146.5 GFLOPS (float32). That's a pretty fast speed. The plane sweep algorithm is firstly designed in a paper in 2008. It was objected to do the real-time online rendering using GPU. In the original paper, three scientists used 11 cameras with a resolution of 320 * 240 and finally got a running speed of 15 frames per second. My kernel used 5 input images with a resolution of 3072 * 2048 (approximately 4k) and got a result of 56 milliseconds(about 20 FPS in theory). The better result may not because of the better optimization I used, but mainly because of the hardware developement within these 12 years. The best graphic card in 2008 is GTX280, and the largest FLOPS of the GTX280 is about 1/11 of my card. But my kernel performance is 50 times of theirs. I didn't find their CUDA kernel code so I really don't know what made the difference between our kernel performance.

IV. Further Optimizing

A. Other Similarity Functions

With the development of computer visions, there's more and more algorithms to find do the same task. Some papers offered a better similarity function to get a better estimation accuracy. The similarity function I used in this project is SAD (sum of absolute difference). So with SAD, you will have a better similarity if the pixels have the same color. But from the basic knowledge of CV, we know that an object's surface may not reflecting the same color towards different directions. So SAD may have a very low accuracy when modeling a surface that do little diffusing reflection. SAD also doesn't work well when we trying to use picture of a same object with different color environment light on it. I've read some of the new techniques nowadays, they all provide a very complex way to calculate the similarity. But the cons of these new methods needs to sacrifice the running speed of the CUDA kernel.

There's also algorithms focusing on optimizing the projection. SAD used pixel to pixel projection, which means find the pixel projected on the input source image. But some algorithms used pixel to window projection. Each real world point on the plane projects to a window on the source input image. If we use this pixel to window projection, we can try to accelerating the kernel using shared memory, since neary by pixels in one thread block are using the same data from global memory. However, there's some difficulty using shared memories. The first one is because of the window size. For example, my CUDA device can only store 49152 bytes per block, which means for each thread, it can only store 14 float numbers on average. But to a large resolution picture, the window size needed is usually above 100. Shared memories usually can not hold that much data. The second reason is the shape of the shared data. It's not shaped squared like what we did in our homework. Usually it is shaped parallelogram. So the algorithm to transfer data from global into shared is quite complex.

There's also algorithms focusing on accelerating performance on sequential code. Some of them are really interesting. A very popular technique is using propagation. The idea is like this. Firstly initialize the estimated depth with random value. Within those random estimation, there must be some

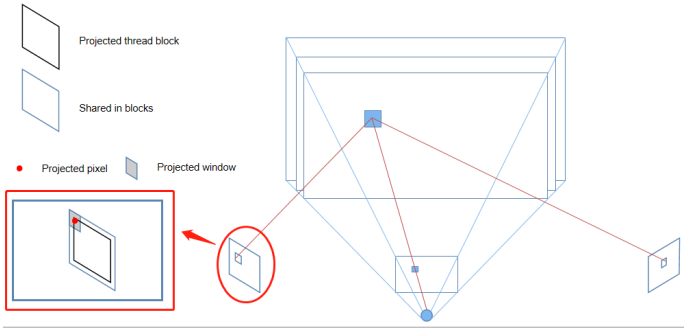

Fig. 14. Pixel to Window Projection

of the values estimated correctly. Then we use the correct value to estimate the pixels around it using the idea that nearby pixels may have the similar depth. And this process is called propagation. We can propagate from left to right on the odd-th propagation and from top to bottom on the even-th propagation. After 4 or 5 propagation, the result will be good as hell. This method is quite smart but not very suitable for parallel programming since there's connections between threads. But it's not executable for GPU just needs a little work on the algorithm design.

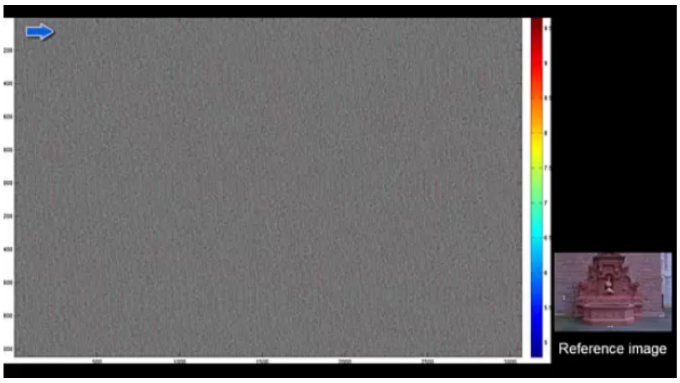

Fig. 15. Pixel to Window Projection

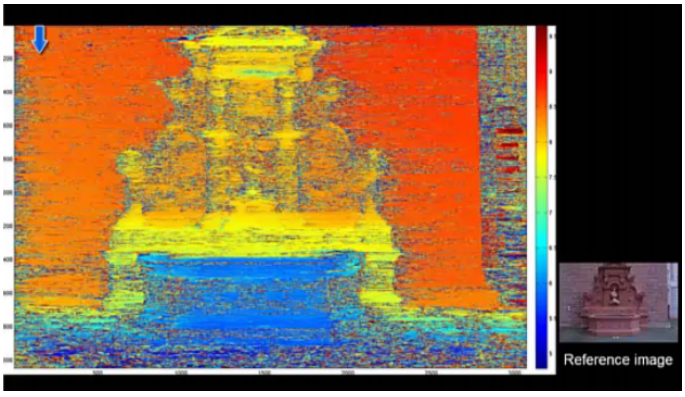

Fig. 16. Pixel to Window Projection

V. Conclusion

I learned and implemented the plane sweep algorithm, using multi-view picture to generate depth map of one camera angle. Firstly, I wrote sequential code in Python to test the algorithm and tune the parameters. After finding the best performance parameter, I used CUDA to write the GPU version of the algorithm. The running speed comparison of the Python sequential code and CUDA parallel code is as following table.

TABLE I
Running Speed Comparison

Language	Resolution	Running Time
Python	307 * 205	10 min
CUDA C	3072 * 2048	55 ms

References

[1] Shi, Bowen Shi, Shan Wu, Junhua Chen, Musheng. (2019). A New Basic Correlation Measurement for Stereo Matching. 10.31219/osf.io/jqux2.

[2] S. Shen, "Accurate Multiple View 3D Reconstruction Using Patch-Based Stereo for Large-Scale Scenes," in IEEE Transactions on Image Processing, vol. 22, no. 5, pp. 1901-1914, May 2013, doi: 10.1109/TIP.2013.2237921.

[3] S. Suwajanakorn, C. Hernandez and S. M. Seitz, "Depth from focus with your mobile phone," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 3497-3506, doi: 10.1109/CVPR.2015.7298972.

[4] Malik, A., Choi, T. (2008). A novel algorithm for estimation of depth map using image focus for 3D shape recovery in the presence of noise. Pattern Recognit., 41, 2200-2225.

[5] Bleyer, Michael Rhemann, Christoph Rother, Carsten. (2011). Patch-Match Stereo - Stereo Matching with Slanted Support Windows. BMVC. 11. 14.1-14.11. 10.5244/C.25.14.

[6] Vincent, Nozick et al. "Plane-Sweep Algorithm: Various Tools for Computer Vision." (2008).

[7] Eigen, David Puhrsch, Christian Fergus, Rob. (2014). Depth Map Prediction from a Single Image using a Multi-Scale Deep Network. 3.

[8] E. Zheng, E. Dunn, V. Jojic and J. Frahm, "PatchMatch Based Joint View Selection and Depthmap Estimation," 2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, 2014, pp. 1510-1517, doi: 10.1109/CVPR.2014.196.