Final Project Report

Kolton Speer

Introduction:

This final project implements a 3D surface reconstruction of bone in CT scans on the GPU using CUDA. Anonymized CT scans were obtained from an online data set provided by qure.ai [1] and an open source repository of CT scans called embodi3D [2]. A 3D model was created by thresholding the images to approximate an isofunction of the bone in the scan and then creating a mesh using the Marching Cubes algorithm. A description of the data, the algorithm, the results, and a runtime analysis are all given below.

Data and Preprocessing:

The data was obtained from open sourced online datasets. Data was run through a python preprocessing script that takes a single DICOM [3] or NRRD[4] file and produces a PGM file for all of the slices. This step was taken to reduce the difficulty of loading the data into the code. Loading data as PGM files reduces the code overhead, but does not change the runtime analysis because the data comes into the code as a 3D matrix of slices either way. The only processing that the data underwent in preprocessing was normalizing pixel values to a range of 0 – 255 and saving as PGM.

Algorithm:

The Marching Cubes algorithm reconstructs a surface by sliding a cube over a 3D data space and determining what combination of triangles to produce on the mesh by looking at what combination of vertices in the cube are inside or outside of the surface. A function known as an isofunction is used to determine which points are contained within the volume of the surface and which points are outside of the surface. An isofunction produces an output of one when the point is completely inside the surface and 0 when it is completely outside the surface. In many applications the output of the isofunction can be viewed as a probability that the given point is inside or outside of the surface.

In this project, an isofunction was approximated by thresholding the input images. This method works really well for CT scans because the bone comes out as being the brightest part of the image. Thresholding was implemented as a GPU kernel since it can be done for every pixel at once and the images have to be loaded onto the GPU anyway. The points above the threshold are set to 255 and the points below are set to zero. In the timing analysis these images are not copied off of the GPU before the marching cubes kernel, but an example of what they look like before being processed by the marching cubes kernel is shown below.

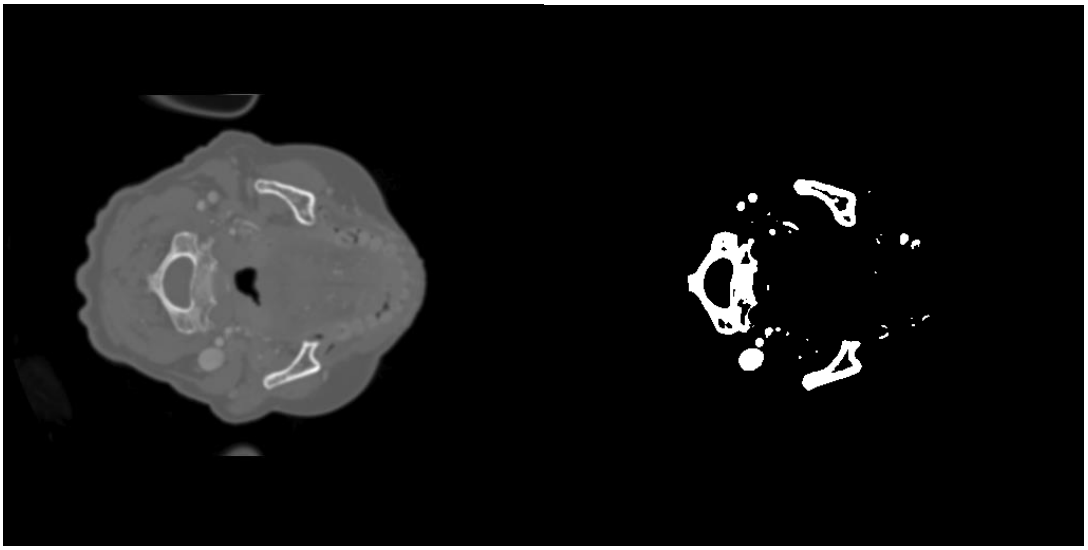


Figure 1 Before and after thresholding

After the thresholding is complete, the pixel values are either 0 or 255 which approximates an isofunction for the surface we're trying to model. The Marching Cubes kernel can then operate over that data using the pixel values as values for the output of the isofunction. Marching Cubes produces triangles by adding the midpoint of an edge of the cube to triangle list if one vertex of the edge is within the surface and one is outside. Since there are a finite number of combinations of vertices on a cube, there are a finite number of possible triangles that can be produced within the cube. If rotations are taken out of the equation, the number of possible triangles in a cube reduces to 15.

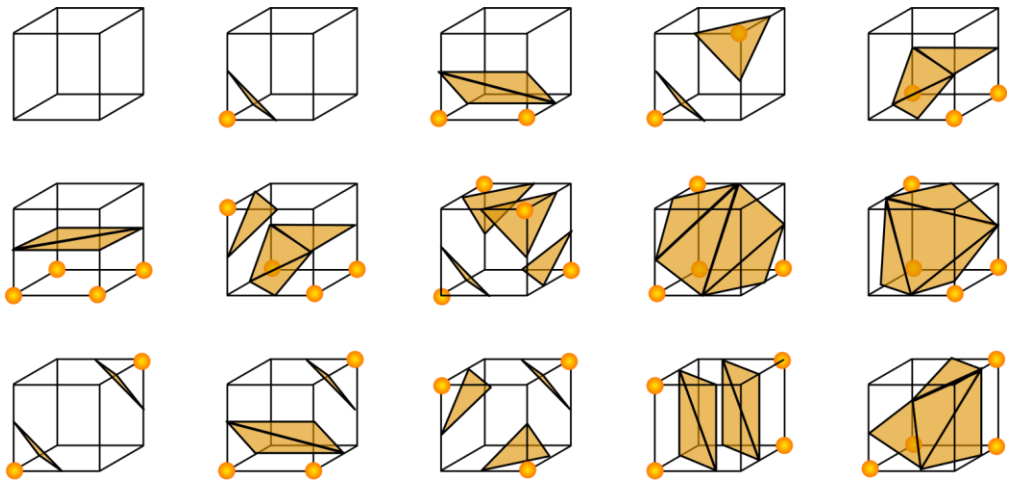


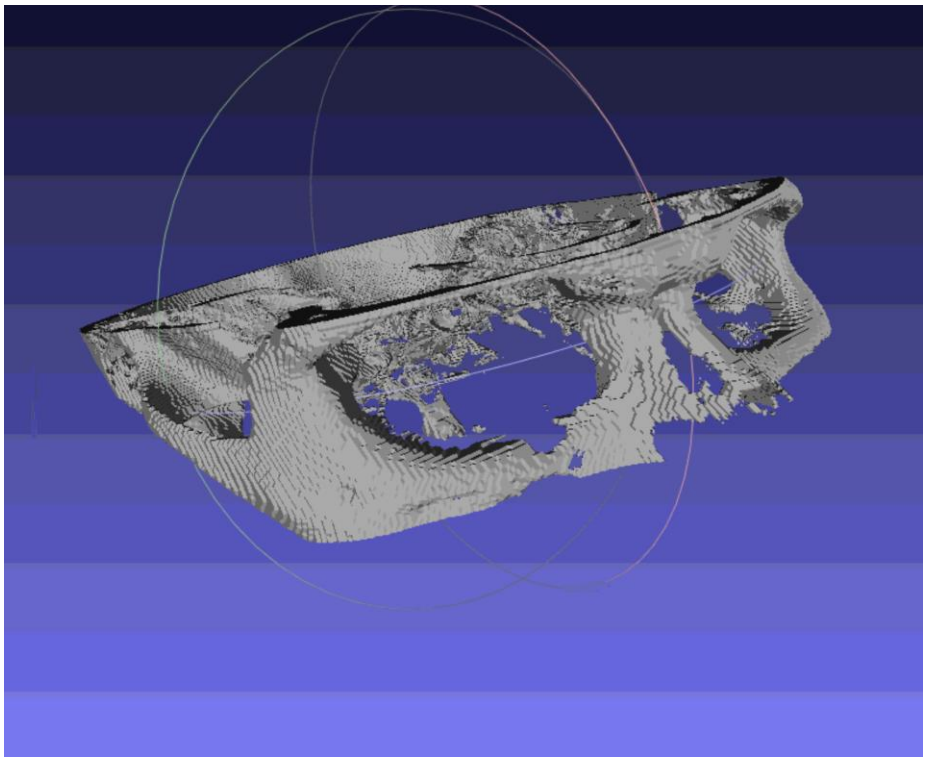
Figure 2 The possible combination of triangles in the cube [5]

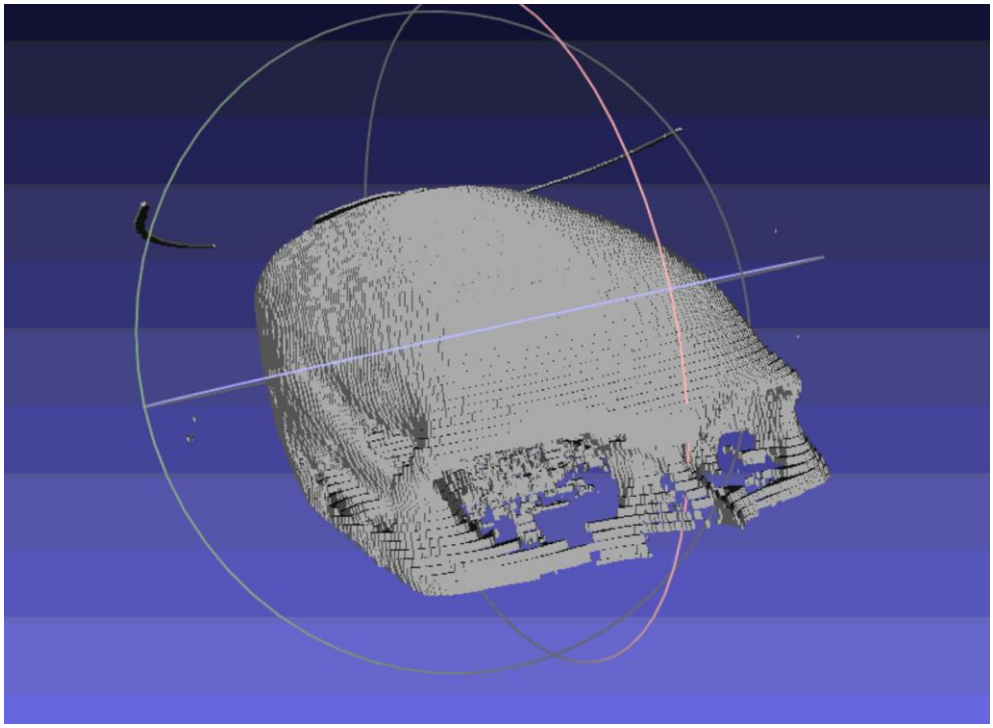
Cube rotations make the number of possible triangles much higher. Calculating the number of combinations is a non-trivial operation, so the possibilities are precomputed and put

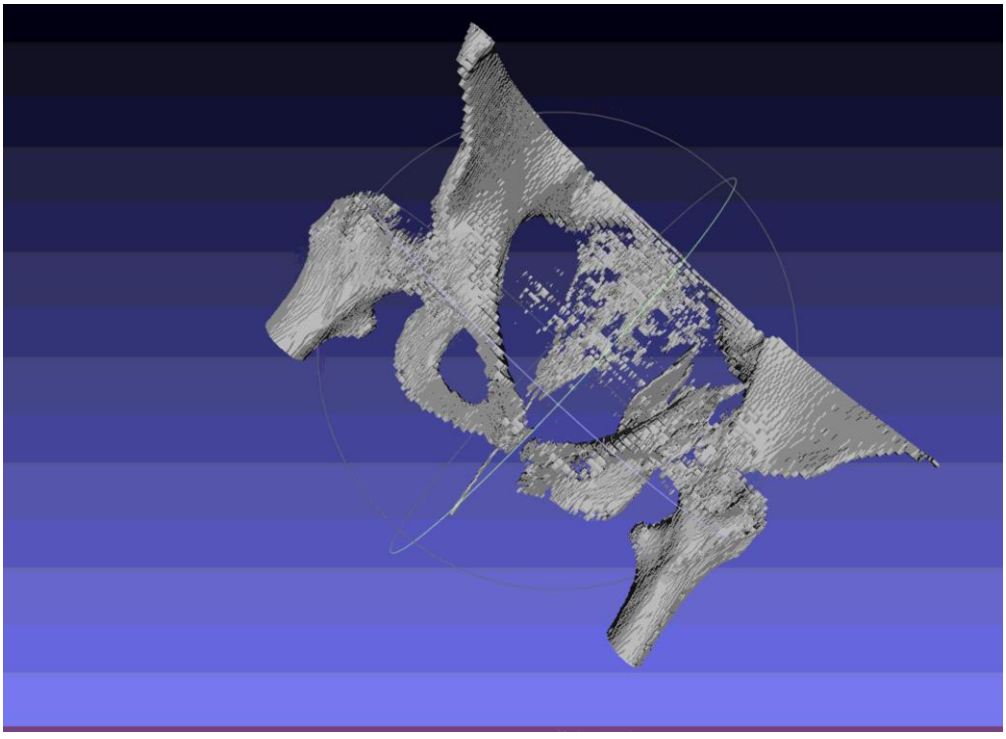
into a lookup table. At every pixel in the matrix the 8 neighboring pixels are taken and the actual points for the triangles are calculated based on the edge combinations from the lookup table. The index in the lookup table is determined by looking at which vertices of the cube are inside or outside of the surface. This is a highly parallelizable algorithm because the lookup can be performed for every pixel in parallel.

Results:

The results of the Marching Cubes kernel written to a PLY file that can be viewed using meshlab or any other PLY file viewer. The results are fairly high fidelity 3D models. A few examples are shown below. The GPU can only handle a certain amount of triangles at a time, so the code is limited to operating on about 62 slices of a scan. Which slices are used is controlled by changing the step size. Consecutive slices will produce a higher fidelity model but of a smaller section of the total scan. Two segments of a skull and a pelvis are shown below.







Runtime Analysis:

The GPU implementation of Marching Cubes is almost entirely limited by copying memory off of the GPU. A quick look at nvprof output on the pelvis scan shows this.

```
==8183== Profiling application: ./kpsgf7_final_project 150 pelvis 62 4 output.ply
==8183== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   85.98%  1.27764s         1  1.27764s  1.27764s  1.27764s  [CUDA memcpy DtoH]
                   13.23%  196.54ms         1  196.54ms  196.54ms  196.54ms  marching_cubes_filter(unsig
                    0.74%  10.959ms         2  5.4797ms  2.8800us  10.957ms  [CUDA memcpy HtoD]
                    0.06%  918.58us         1  918.58us  918.58us  918.58us  thresholding_filter_kernel(
      API calls:   66.11%  1.28968s         3  429.89ms  19.486us  1.27853s  cudaMemcpy
                   23.06%  449.74ms         2  224.87ms  2.9700us  449.73ms  cudaEventCreate
                   10.12%  197.48ms         2  98.738ms  922.34us  196.55ms  cudaDeviceSynchronize
                    0.36%  6.9756ms         3  2.3252ms  165.34us  4.2942ms  cudaFree
                    0.29%  5.7168ms         5  1.1434ms  4.9870us  4.6984ms  cudaMalloc
                    0.03%  590.79us         4  147.70us  144.19us  151.19us  cudaGetDeviceProperties
                    0.01%  204.46us         1  204.46us  204.46us  204.46us  cuDeviceTotalMem
                    0.01%  169.20us        94  1.8000us     176ns  70.139us  cuDeviceGetAttribute
                    0.00%  60.564us         2  30.282us  16.347us  44.217us  cudaLaunch
                    0.00%  25.995us         1  25.995us  25.995us  25.995us  cuDeviceGetName
                    0.00%  23.878us         2  11.939us  7.7790us  16.099us  cudaEventRecord
                    0.00%  4.4600us         1  4.4600us  4.4600us  4.4600us  cudaSetDevice
                    0.00%  4.2100us         1  4.2100us  4.2100us  4.2100us  cudaEventSynchronize
                    0.00%  3.3080us         2  1.6540us     505ns  2.8030us  cudaConfigureCall
                    0.00%  3.0580us         3  1.0190us     277ns  2.3800us  cuDeviceGetCount
                    0.00%  2.6170us        11     237ns     109ns     560ns  cudaSetupArgument
                    0.00%  1.9210us         1  1.9210us  1.9210us  1.9210us  cudaEventElapsedTime
                    0.00%  1.3430us         1  1.3430us  1.3430us  1.3430us  cudaGetDeviceCount
                    0.00%  1.2740us         2     637ns     343ns     931ns  cudaPeekAtLastError
                    0.00%  1.0900us         2     545ns     214ns     876ns  cuDeviceGet
                    0.00%     682ns         2     341ns     255ns     427ns  cudaGetErrorString
```

The nvprof output shows us that the GPU spends almost 86% of its entire operating time performing copy operations. The total time for the thresholding kernel and the marching cubes kernel in this example totals out to about 200 ms while the entire GPU operation totals to 1493.28 ms. The CPU operation on the same data takes 3399.17 ms. These numbers are very similar to numbers on other scans.

Conclusion:

The sheer amount of memory needed to store and copy the full amount of triangles is the single most limiting factor in this project. A more efficient method of storing triangles is definitely needed in order to make this faster and more useful. However, the project does show that the Marching Cubes algorithm can run extremely fast on the GPU and beats CPU times even with the copying operations involved.

Works Cited

[1] Chilamkurthy, S. (2018, April 12). Qure.ai. Retrieved December 13, 2019, from http://headctstudy.qure.ai/#dataset

[2] Embodi3D. (2019, December 13). Retrieved December 13, 2019, from https://www.embodi3d.com/

[3] DICOM Standard. (2019, December 13). Retrieved December 13, 2019, from https://www.dicomstandard.org/

[4] Teem: Definiton of NRRD File. (2019, December 13). Retrieved December 13, 2019, from http://teem.sourceforge.net/nrrd/format.html

[5] Marching Cubes. (2019, November 12). Retrieved December 13, 2019, from https://en.wikipedia.org/wiki/Marching_cubes