Engraver: GPU and CPU Parallel Implementation of Edge Detector

Kaige Liu, Karen He

Summary

We implemented from scratch two parallel versions of the Canny edge detector, a CUDA version on GPU and an OpenMP version on CPU, and compared the performance of the two implementations. We optimized our implementation to achieve a 8 times speedup with our GPU implementation and a 6 times speedup with our CPU implementation using 8 cores.

Keywords: Canny edge detection, image segmentation, CUDA, OpenMP.

Project URL: https://dukaige.github.io/kaigel_rhe1_418project/

1 Background

Edge detection is an image processing technique for finding the boundaries of objects within images. It works by identifying points in a digital image at which the image brightness changes sharply or, more formally, has discontinuities. Edge detection has a wide variety of application, including image segmentation and data extraction, in many areas such as image processing, computer vision, and machine vision.

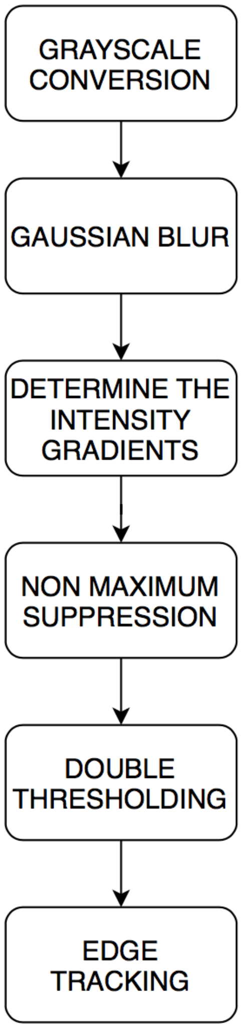


Figure 1: Edge Detection Operations [?]

In our project, we implemented an edge detector based on the Canny edge detector. We also explored the application of edge detection by implementing an image segmentation algorithm that allows us to achieve image cutout.

Our program is divided into two parts, the edge detector, and the image segmentor. The input of the edge detector is an image, and the output is the edge extraction image of the input image. We represent the image as a vector of float throughout the process. For the image vector, we present a series of operations as illustrated in Figure 1.

1.1 Greyscale and Blur

We first convert the image into greyscale to simplify the problem. Since our objective is to extract the edges, the presence of multiple channels is a distracting factor. Then we perform a Gaussian blur on the image. The blur removes some of the noise before further processing the image.

1.2 Extract Gradient

The third step is to extract the gradients of the image, in order to detect the edges. An edge occurs when the color of an image changes, hence the intensity of the pixel changes as well. We calculate the magnitude and angle of the directional gradients, which can be determined by using a Sobel filter.

1.3 Non-Maximum Suppression

The next step is non-maximum suppression, i.e. edge thinning. The image magnitude produced results in thick edges. Ideally, the final image should have thin edges. Thus, we perform non-maximum suppression to thin out the edges. Non-maximum suppression works by finding the pixel with the maximum value in an edge. As illustrated in Figure 2, it occurs when pixel q has an intensity that is larger than both p and r where pixels p and r are the pixels in the gradient direction of q. If this condition is true, then we keep the pixel, otherwise, we set the pixel to zero (make it a black pixel).

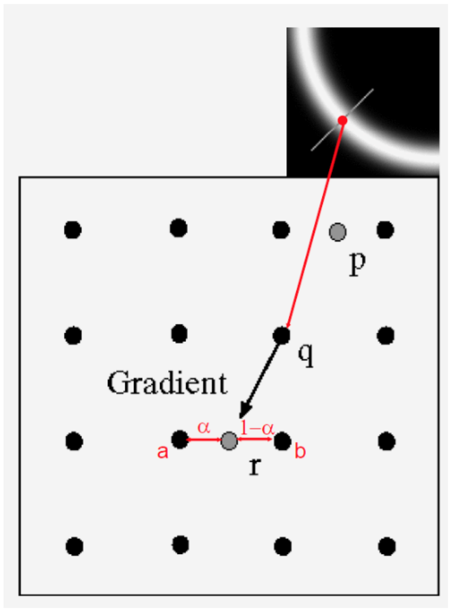


Figure 2: Non-Maximum Suppression

1.4 Double Thresholding

The fifth step is double thresholding. We notice that the result from non-maximum suppression is not perfect, since some edges may not actually be edges due to some noise in the image. Double thresholding takes care of this. It sets two thresholds, a high and a low threshold. Pixels with a high value are most likely to be edges. We normalized all the values such that they will only range from 0 to 1. All pixels with a value larger than the high threshold will be classified as a strong edge. All pixels with a value smaller than the high threshold will be set to 0. The values between the low threshold and the high threshold would be classified as weak edges, which we do not know whether they are actual edges or not edges at all. The two thresholds are different per image, based on the characteristics of the image. We found it helpful to choose a threshold ratio instead of a specific value and scale it by multiplying the maximum pixel value in the image. As for the low threshold, we chose a low threshold ratio and multiplied it by the high threshold value.

1.5 Edge Tracking

The last step is edge tracking. Now that we have determined what the strong edges and weak edges are, we need to figure out which weak edges are actual edges. To do this, we perform edge tracking by doing a graph search. Weak edges that are connected to strong edges (including those strong edges that are formed by other weak edges) will be classified as real edges. Weak edges that are not connected to strong edges will be removed. In our sequential version, we implement this expansion process using depth-first search.

1.6 Parallelism

The computationally expensive part of the edge detection lies in the fact that, in each step, we have to do work on every single pixel of the image. When the image gets large, each step can take a huge amount of time. Therefore, the edge detection significantly benefits from parallelism, especially for large input images.

Every step is dependent on the result of the previous step, so we have to insert barriers between every two steps. For steps 1-5, the internal dependency within the step is low, since the work of each pixel is not dependent on other pixels. Therefore we fully parallelled the work on each pixel. For step 6, as we are constantly expanding from the strong edges to infect there neighboring weak edges, the work done on each pixel is dependent on its neighbors, neighbors of neighbors, and so on. Therefore we cannot take a data-parallel approach. However, we can still take advantage of spatial and temporal locality given we are expanding from strong edges to its neighbors, and neighbors of neighbors, and so on.

2 Approach

After implementing a sequential edge detector, we implemented two parallel versions of the edge detector — a GPU version using CUDA, and a CPU version using OpenMP. To obtain a good performance on the CUDA version, we used the GHC machines that contain NVIDIA GeForce GTX 1080 GPUs, which support CUDA compute capability 6.1. We chose to use one of the GHC machines since it has a very powerful graphics card, and it also provides a reasonable processor for comparison. We believe a GHC machine reflects the computational abilities of many modern machines.

2.1 CUDA Implementation on GPU

2.1.1 Our Parallelization Techniques

For steps 1-5, where the work is independent, we directly map the pixels to threads. For step 6 (edge tracking), we present a simulation of the depth-first search. This is the only place that we change the original sequential version to compromise accuracy for speedup. We first divide the image into a number of blocks. We make the number of blocks large (the number of GPU CUDA threads $\times$ 10), and leave the assignment of CUDA wraps to blocks in CUDA scheduler.

After dividing the image, we perform a number of iterations to approximate the result of the actual depth-first search. In each iteration, we perform depth-first search within each block and exchange the values of the pixels at the border with their neighbors across the block. An illustration is shown in

Figure 3, where the pixels whose value need to be exchanged are marked blue.

After experimenting with different image and different input size, we discovered that for most images, the result after two to three iterations would closely approximate the actual depth-first search result. In fact, after only one iteration, the result is roughly the same as the actual depth-first search result.

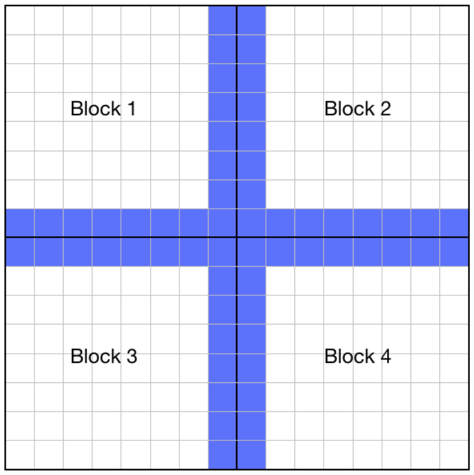


Figure 3: Easy Input Image

This is within our expectation, given the nature of the task. In the edge tracking step, we want to set the weak edges closed to strong edges to be strong edges. Among the pixels that are not set to zero after non-maximum suppression, the majority is strong edges. The remaining small portion of weak edges typically consists of two parts: the unclear boundaries, which are usually close to the strong edges, and noises, which are often far apart from the actual (strong) edges. Therefore, we dont need many iterations to approximate the actual DFS result. We ended up performing two iterations of DFS, given that DFS is the most computationally expensive part of the algorithm, and that the accuracy wont be compromised much.

2.1.2 Other Attempts

Before arriving at this DFS approximation algorithm, we had some failed attempts. We initially planned to use the red-black coloring discussed in class to divide the pixels into two groups and alternate updating the two groups. A problem with this approach is that we have to synchronize between every two updates, while each update is relatively computationally cheap. Thus the communication ratio is high. Moreover, this approach requires a large number of iterations (updates), within which the work done on each pixel gets increasingly unbalanced. Thus we discarded this approach. Another attempt of improvement is to decide the number of iterations in the block DFS approach dynamically, i.e. counting the ratio of useful exchanges across the boundary to decide whether we should perform another iteration. However, this approach causes unnecessary contention since all the wraps are counting the number of useful exchanges simultaneously. Thus we also discarded this approach and chose to perform a fixed number of iterations.

2.2 OpenMP Implementation on CPU

2.2.1 Our Parallelization Techniques

Similar to the GPU version, for steps 1-5, where the work is independent, we directly map the pixels to threads. We use static scheduling since the work done on each pixel is fixed and predictable.

To divide the image into blocks, we adopt the block assignment. Initially, we used the interleaved assignment, i.e. assigning a row of pixels to a thread. However, this would result in frequent cache invalidation. For example, in the gradient extraction step, each pixel has to refer to the values of its neighbors. For a pixel at the border of a block, while it writes to itself, its neighbor across the block accesses its value. The contention between reading and writing can result in unnecessary cache invalidation. Thus we turned to block assignment to reduce the ratio of pixels near the border of the blocks.

For step 6 (edge tracking), we again present a simulation of the depth-first search using the same block DFS approach as the GPU version. What is different is how we map threads to pixels. In OpenMP version, the number of blocks we divide the image into is 20 × the number of processors. We then use dynamic scheduling to assign blocks to threads, since the work done on each block is unpredictable and constantly changing. After dividing the image, we use the same techniques as the GPU version to simulate DFS. What is different is that we use MPI_Send and MPI_Recv to exchange the values of the pixels at the border with their neighbors across the block.

2.3 Image Segmentation

After finishing our project goal, the two parallel versions of the edge detector, we explored its application by implementing an image segment. The input is the edge extraction graph and a point inside the image. The output is an image with only the segmented portion of a graph that contains the designated point. We can then use this as the mask to filter out this portion of the image to reach the effect of object cutout. An illustration of the object cutout effect based on different designated points on the same image is illustrated in Figure 5.

We implemented this part by radiating from the designated point to find a point on its

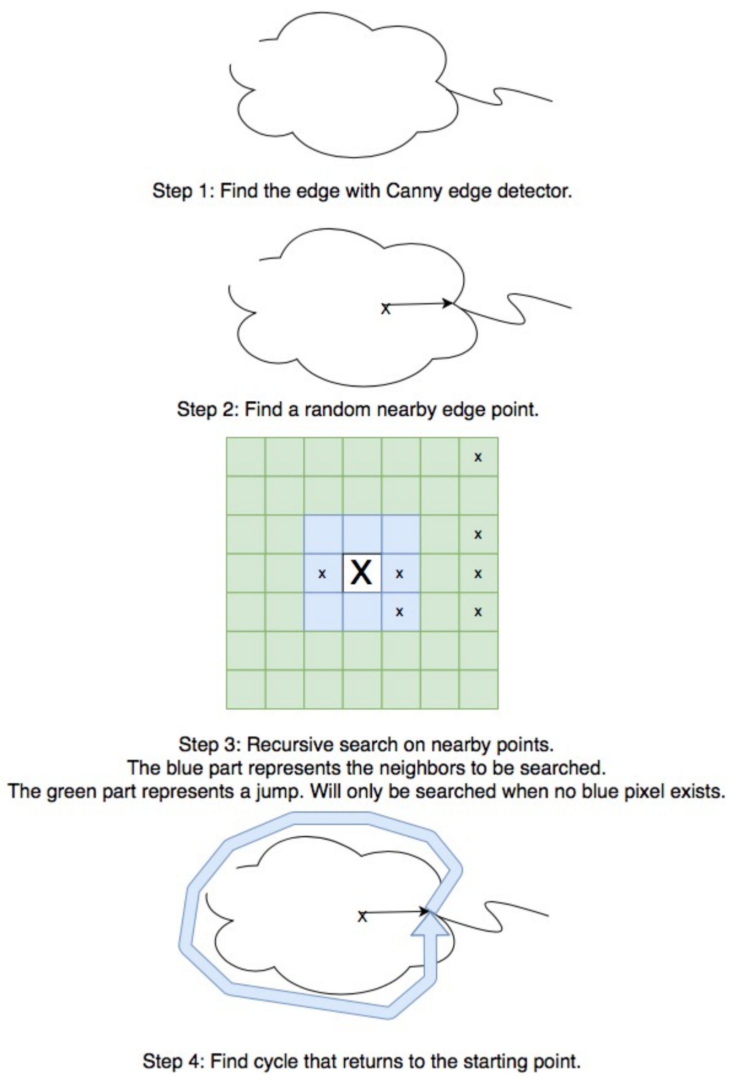

Step 1: Find the edge with Canny edge detector.
Step 2: Find a random nearby edge point.
Step 3: Recursive search on nearby points.
The blue part represents the neighbors to be searched.
The green part represents a jump. Will only be searched when no blue pixel exists.
Step 4: Find cycle that returns to the starting point.

Figure 4: Image Segmentation Procedure

boundary and following along that point to find a closed boundary that encloses the designated point. Whenever we fail in the process, such as hitting the image border, failing to find a closed boundary, or finding a boundary that does not enclose the designated point, we start over the search. After finding the correct boundary, we suppress all other edges and output only the boundary. An illustration of the imgae segmentation algorithm is shown in Figure 4.

All parts of our program were started from

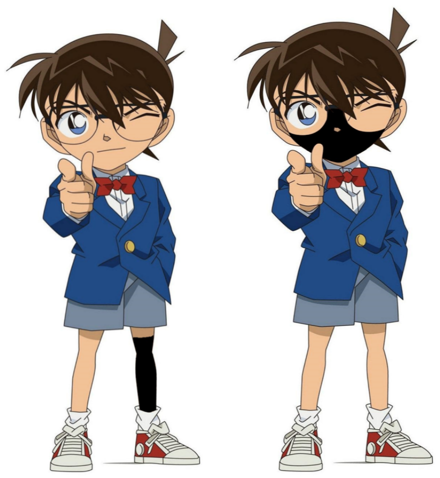


Figure 5: Image Segmentation Example

scratch, without any code base or data processing platforms such as OpenCV.

3 Results

3.1 Experimental Setup

We performed extensive testing on both our GPU and CPU parallel implementations. We utilized the machines on the 5th floor of GHC.

For each of the parallel version, we experiment with four different input configurations. We chose an easy image, with clear and continuous edges, as shown in Figure 6. We also chose three relatively hard images (real photos with not necessarily clear boundaries) with different sizes: 2k (Figure 12), 4k (Figure 13) and 8k (Figure 14). The edge detection results of these three images are shown in the appendix.

We measure our performance by measuring the time elapsed for each part of the algorithm. We excluded the time of image reading and file writing, and only measured the parts involving edge detection.

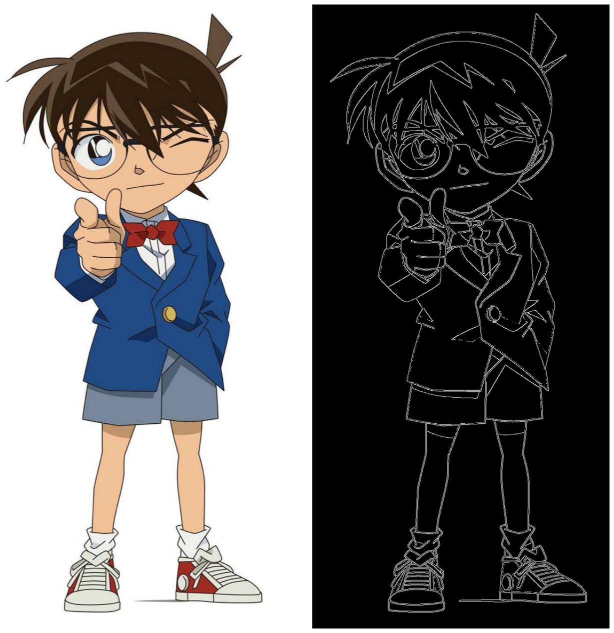


Figure 6: Easy Input Image

3.2 GPU Performance and Analysis

For the GPU version, we varied the problem size and measured the speedup relative to the CPU version with 1 processor. The result is shown in Table 1 attached in Appendix, and a line graph is shown in Figure 7.

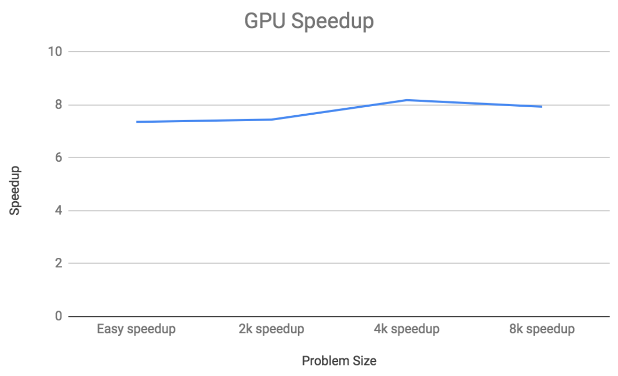


Figure 7: GPU Speedup Graph

As shown in Figure 7, the speedup of our GPU implementation is constantly 7 to 8 times compared to the runtime of our baseline sequential implementation on CPU. The speedup is almost independent of the problem size.

Although a 8 to 9 times speedup is satisfactory, the speedup doesn't fully reflect the extra parallel computational resources provided on a GPU. To determine the exact reason, we broke the execution time of our algorithm into a number of distinct components, according to the steps described in the Background section. The result is shown in Table 3 attached in Appendix, and a line graph is shown in Figure 8.

As for edge tracking, we believe one of the limitations is the low SIMD utilization in our CUDA implementation. We implemented edge tracking with the DFS algorithm, and we used a stack-like data structure to keep track of the process of DFS, which highly diverges due to the difference in the number of iterations and conditionals.

The other components all experience very high speedup, since the operations are less memory intensive, and experience very low SIMD divergence. According to our measurements, each component experience roughly 50-100x speedup.

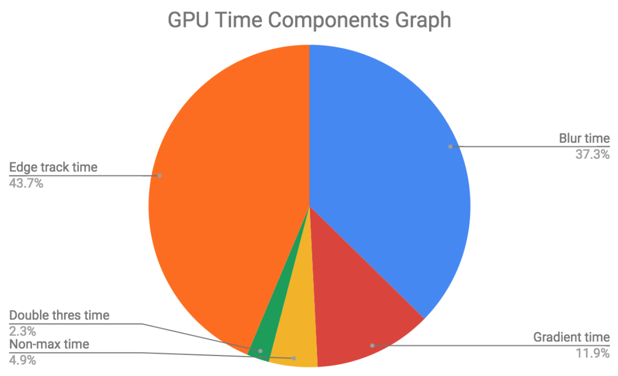


Figure 8: GPU Time Components Graph

According to Figure 8, the most expensive parts in our GPU implementation are Gaussian blur (37.5%) and edge tracking (43.7%). In the Gaussian blur process, each unit reads data from its neighboring pixels, and calculates a weighted sum according to the Sobel filter. Such operation produces a high traffic for the memory bandwidth. Since the task is memory bounded, the performance of Gaussian Blur is bounded by memory bandwidth, and therefore less dependent on GPU's extra computational power. The same problem applies to gradient calculation, which occupies 11% of the total time. Gradient calculation requires accessing nearby memory blocks, and is therefore tightly bounded by memory bandwidth.

3.3 CPU Performance and Analysis

For the CPU version, we varied both the number of processors and the problem size and measured the speedup relative to the CPU version with 1 processor. The result is shown in Table 2 attached in Appendix, and a line graph is shown in Figure 9.

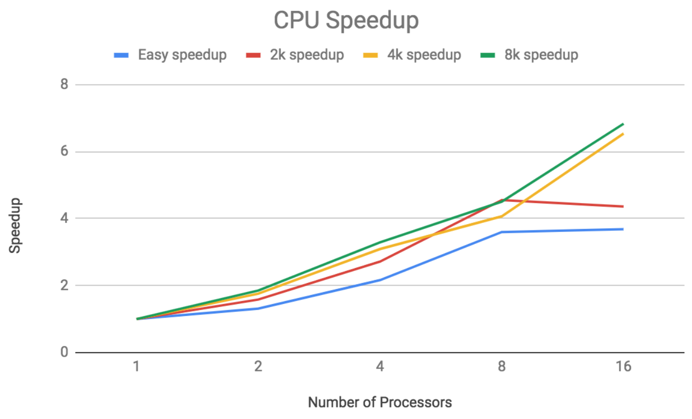
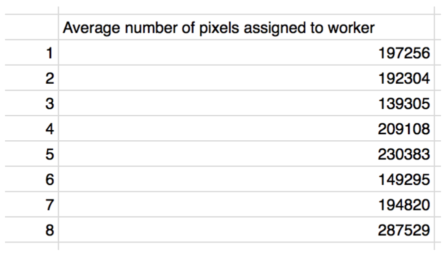


Figure 9: CPU Speedup Graph

According to Figure 11, we generally obtained a 5x speedup with 8 cores and 8x speedup with 16 cores, which is nearly linear with respect to the number of processors. The speedup also greatly varies with different input size. When we experiment with larger images, the CPUs implementation usually experiences a higher speedup. We believe this is due to better work balancing among various workers. To verify our conjecture, we performed measurement on the number of pixels assigned to each processor in edge linking step, shown in 10. The data shows that Unlike our GPU implementation, each computation unit in the CPU is responsible for a large number of pixels, and therefore work balancing becomes a bigger issue. As we mentioned before, a large portion of our CPU implementation is scheduled statically. Statistically, as the number of pixels increases, pixels can be more evenly distributed among workers and therefore fewer workers become idle.



	Average number of pixels assigned to worker
1	197256
2	192304
3	139305
4	209108
5	230383
6	149295
7	194820
8	287529

Figure 10: Number of pixels assigned to each worker

We also noticed that, for smaller problem sizes, the speedup decreases when we increase from 8 scores to 16 cores. We believe this is due to the performance overhead of dynamic work distribution. When the workload is small, the ratio of scheduling time will drastically increase.

To identify more limitations to our CPU implementation, we varied the number of processors and measured the time breakdown for each configuration. The result is shown in Table 4 attached in Appendix, and a line graph is shown in Figure 11.

We identified from the measurements that among all components in the computation, edge tracking and Gaussian Blur experience the most obvious speedup among all components, whereas gradient and non-maximum suppression experience the lowest. We believe this is due to the work division of our implementation. In both gradient calculation and non-maximum suppression, our work assignment will result in each worker reading data from data segment responsible by other workers. Therefore this will cause extra invalidation when workers write across different segments, and extra interconnect traffic when writing the data responsible by other workers.

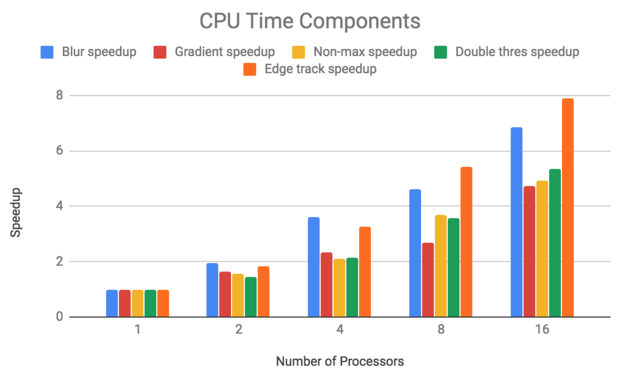


Figure 11: CPU Time Components Graph

4 Conclusion

We implemented parallel edge detection for both GPU with CUDA and CPU with OpenMP. The implementations managed to perform edge detection and produce image segmentation using the result of edge detection. We also performed very extensive measurements. By our measurement, we found out that our implementation has a better performance on NVIDIA GTX 1080 than 8-core Intel Xeon E5, which leads to the conclusion that GPU is a better choice for this parallel task. We also identified the GPU performance is mostly bounded by memory bandwidth, which means that increasing memory bandwidth can potentially improve the speedup of our parallel implementation.

5 References

[1] Green, Bill. "Canny edge detection tutorial." Retrieved: March6 (2002): 2005.
[2] Canny, John. "A computational approach to edge detection." IEEE Transactions on pattern analysis and machine intelligence 6 (1986): 679-698.
[3] Harish, Pawan, and P. J. Narayanan. "Accelerating large graph algorithms on the GPU using CUDA." International conference on high-performance computing. Springer, Berlin, Heidelberg, 2007.
[4] Morar, Anca, Florica Moldoveanu, and Eduard Grller. "Image segmentation based on active contours without edges." 2012 IEEE 8th International Conference on Intelligent Computer Communication and Processing. IEEE, 2012.
[5] Pal, Nikhil R., and Sankar K. Pal. "A review on image segmentation techniques." Pattern recognition 26.9 (1993): 1277-1294.
[6] Felzenszwalb, Pedro F., and Daniel P. Huttenlocher. "Efficient graph-based image segmentation." International journal of computer vision 59.2 (2004): 167-181.
[7] https://github.com/DUKaige/kaigel_rhe1_418project.

6 Work division

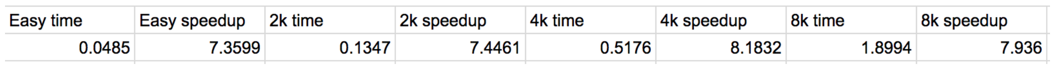
- Sequential edge detector
Dustin: non-maximum suppression, double thresholding
Karen: gradients extraction, edge tracking

- Parallel edge detector on GPU
Dustin: edge tracking
Karen: gradients extraction, non-maximum suppression, double thresholding

- Parallel edge detector on CPU
Dustin: edge tracking
Karen: gradients extraction, non-maximum suppression, double thresholding

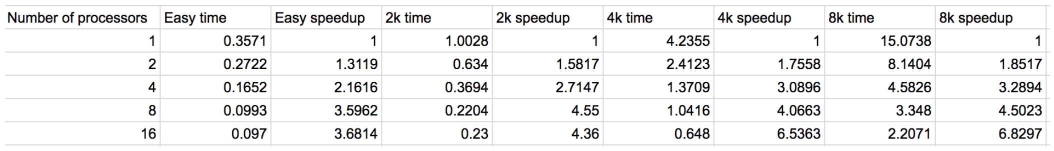
- Image segmentor
Dustin: Point radiation and closed boundary search
Karen: Deciding whether a point is within a closed boundary

We think that the total credit for the project should be distributed 50%, 50% amongst our two.



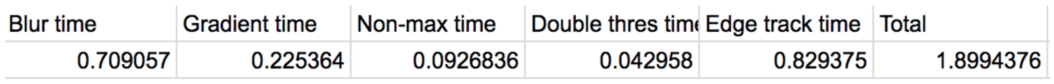
Easy time	Easy speedup	2k time	2k speedup	4k time	4k speedup	8k time	8k speedup
0.0485	7.3599	0.1347	7.4461	0.5176	8.1832	1.8994	7.936

Table 1: GPU Speedup Data



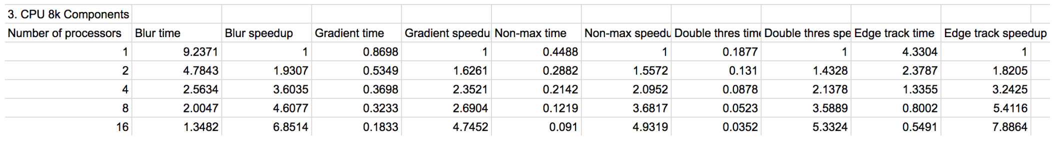
Number of processors	Easy time	Easy speedup	2k time	2k speedup	4k time	4k speedup	8k time	8k speedup
1	0.3571	1	1.0028	1	4.2355	1	15.0738	1
2	0.2722	1.3119	0.634	1.5817	2.4123	1.7558	8.1404	1.8517
4	0.1652	2.1616	0.3694	2.7147	1.3709	3.0896	4.5826	3.2894
8	0.0993	3.5962	0.2204	4.55	1.0416	4.0663	3.348	4.5023
16	0.097	3.6814	0.23	4.36	0.648	6.5363	2.2071	6.8297

Table 2: CPU Speedup Data



Blur time	Gradient time	Non-max time	Double thres tim	Edge track time	Total
0.709057	0.225364	0.0926836	0.042958	0.829375	1.8994376

Table 3: GPU Components Data



3. CPU 8k Components										
Number of processors	Blur time	Blur speedup	Gradient time	Gradient speedu	Non-max time	Non-max speedu	Double thres tim	Double thres spe	Edge track time	Edge track speedup
1	9.2371	1	0.8698	1	0.4488	1	0.1877	1	4.3304	1
2	4.7843	1.9307	0.5349	1.6261	0.2882	1.5572	0.131	1.4328	2.3787	1.8205
4	2.5634	3.6035	0.3698	2.3521	0.2142	2.0952	0.0878	2.1378	1.3355	3.2425
8	2.0047	4.6077	0.3233	2.6904	0.1219	3.6817	0.0523	3.5889	0.8002	5.4116
16	1.3482	6.8514	0.1833	4.7452	0.091	4.9319	0.0352	5.3324	0.5491	7.8864

Table 4: CPU Components Data

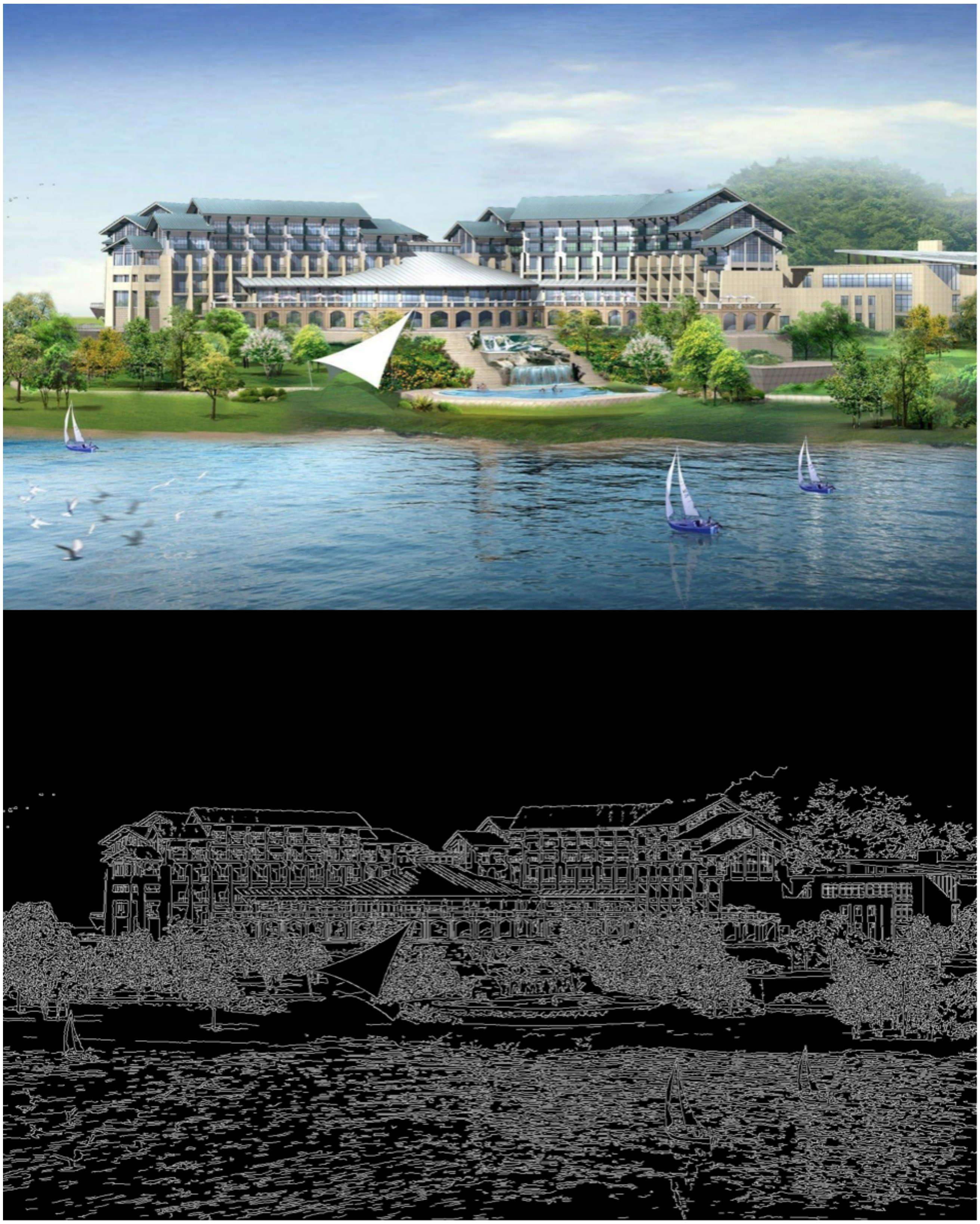


Figure 12: 2K 1600x1000 Input Image

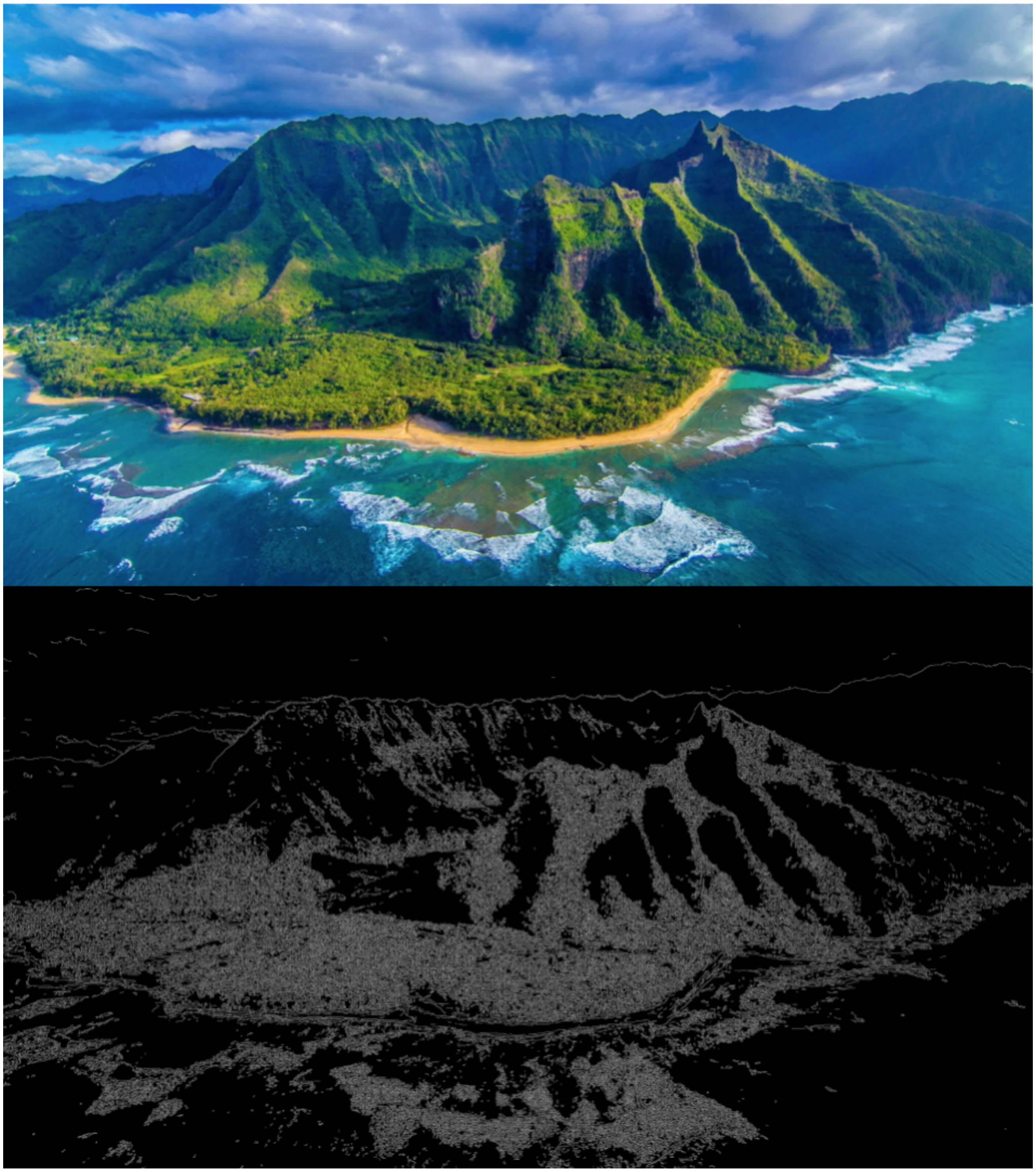


Figure 13: 4K 3840x2160 Input Image

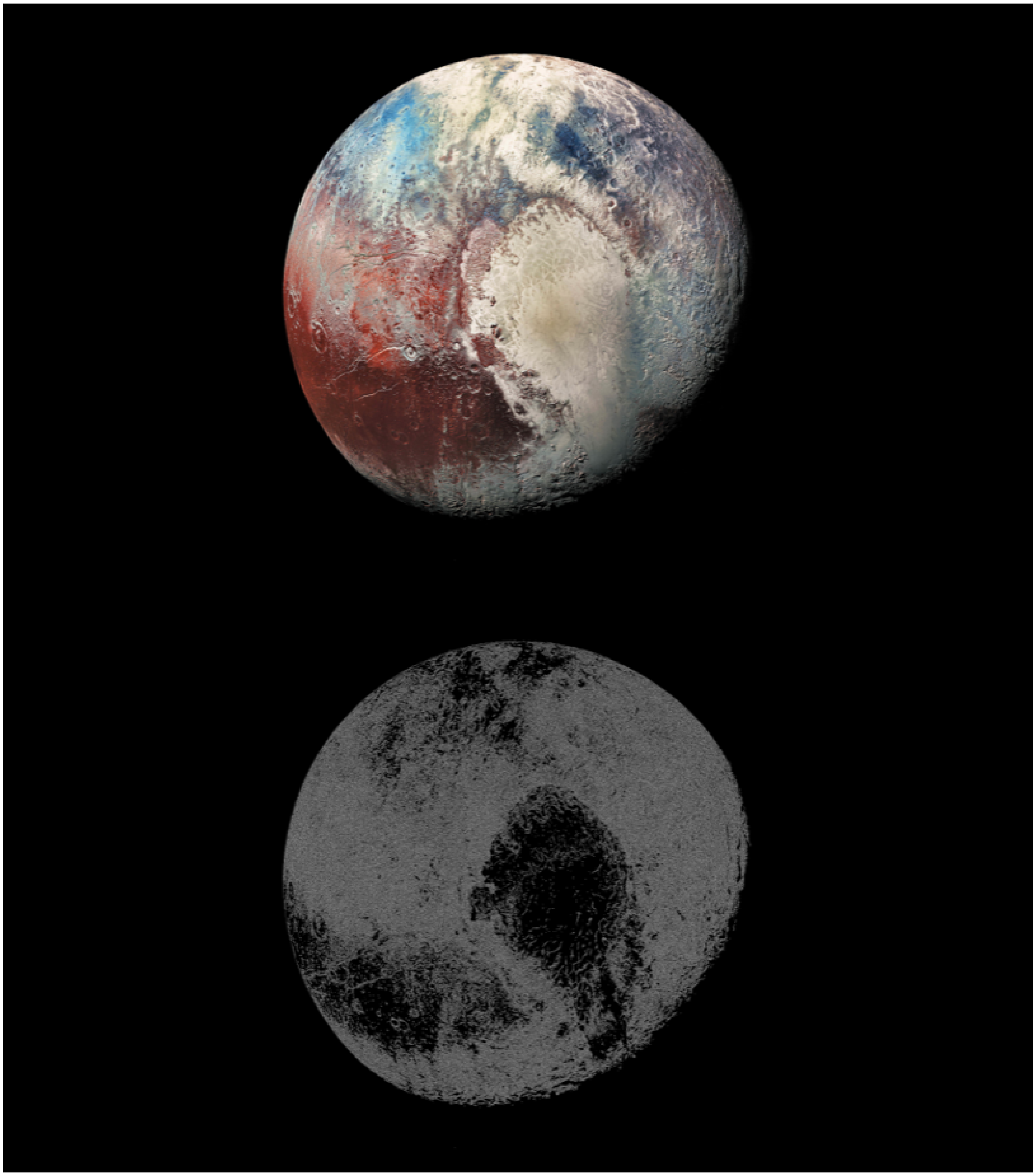


Figure 14: 8K 7500x4300 Input Image