Parallelized Canny Edge Detection

Hong Yu Chiu
National Chiao Tung University
Taiwan

Cheng Chin Tsai
National Chiao Tung University
Taiwan

Yang Chieh Sheng
National Chiao Tung University
Taiwan

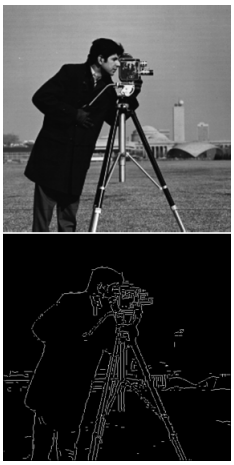

Figure 1: Original image(upper) and final image(lower).

Figure 2: Original image(upper) and after noise reduction(lower).

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**.

KEYWORDS

canny edge detection, parallel programming, sobel operator, cuda

1 ABSTRACT

In the project, we aim to speed up the calculation time of Canny Edge detection which is well-known for representing the focus of the picture with a border. The main library we import is CuPy which is built on Python with CUDA. Not only using a single image to compare accelerated performance, we also use different complexity and different size images to compare performance.

2 INTRODUCTION

The Canny edge detector was developed by John F. Canny in 1986. It is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images [3]. The Process of Canny edge detection algorithm consists of four steps:

- Noise reduction.
- Gradient intensity.
- Non-maximum suppression.
- Double threshold.

After applying these steps, you will be able to get the result such as Figure 1.

2.1 Noise Reduction

All edge detection results are susceptible to image noise, so we must filter out it to prevent false detection. In noise reduction step, we convert the image to grayscale and then convolve the Gaussian filter kernel with the grayscale image to reduce the effects of obvious noise on the edge detector. The kernel size depends on the expected blurring effect. Basically, the smallest the kernel, the less visible is the blur. Equation for a Gaussian filter kernel of size (2k+1)*(2k+1) is given by:

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp \frac{-(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2}; 1 \leq i, j \leq (2k+1) \tag{1}$$

After applying the Gaussian blur, we get the result such as Figure 2:

2.2 Gradient Intensity

The edges correspond to changes in pixel intensity. In gradient intensity step, we use edge detection operator, Sobel filter, get the first derivatives in the horizontal direction (Gx) and the vertical direction (Gy).

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}; G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2}$$

After obtaining two images in the horizontal and vertical directions, the edge gradient size G and slope ϕ of each pixel can be calculated using the formula 3 and 4:

$$G = \sqrt{G_x^2 + G_y^2} \tag{3}$$

$$\phi = \arctan \frac{G_y}{G_x} \tag{4}$$

After applying the gradient calculation, we get the result such as Figure 3:

Figure 3: Gradient intensity.

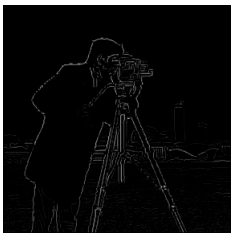

Figure 4: Non-maximum suppression.

2.3 Non-maximum suppression

After step 2, you can see the image is almost the one what we want, but some of the edges are thick and others are thin. Non-Max Suppression step can help us solves this problem. Linear interpolation is used between the two neighboring pixels that straddle the gradient direction. The gradient magnitude at the central pixel must be greater than both of these. If the condition holds, the central pixel will be marked as an edge, otherwise it will be replaced by 0. Algorithm goes through all the points on the gradient intensity matrix and finds the pixels with the maximum value in the edge directions. In other words, based on edge direction and pixel intensity, for each pixel the Non-Max Suppression steps can be divided into four steps:

- Create a zero-matrix of the same size as the original gradient intensity matrix.
- Identify the edge direction based on the angle value from the angle matrix.
- Checking if the pixel in the same direction has a higher intensity than the pixel that is currently processed.
- Returning the image processed with the non-max suppression algorithm [5]

After applying the gradient calculation, result is the same image with thinner edges. It is like Figure 4:

2.4 Double threshold

Although we found the real boundary, as can be seen from Figure 4, some points on the border are brighter and some are darker. To solve this problem, we first need to set two thresholds, which are high threshold and low threshold. The double thresholds will help us classify pixels into three types: strong, weak, and non-relevant.

- Strong: Pixel gradient intensity greater than high threshold.
- Weak: Pixel gradient intensity smaller than high threshold and greater than low threshold.

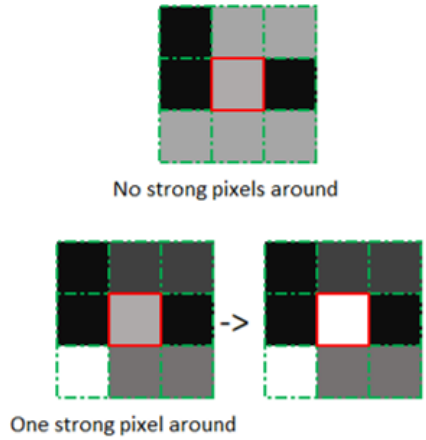

Figure 5: Classification rules.

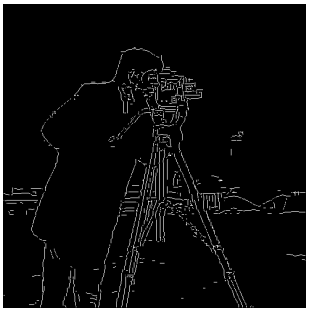

Figure 6: Final image.

- Non-relevant: Pixel gradient intensity smaller than low threshold.

In order for the boundaries to connect without breaking, weak boundaries are necessary. But our ultimate goal is only two categories, boundary or not, so we need to deal with weak boundaries again. If and only if at least one of the pixels around the one which being processed is strong, the pixel being processed is considered as a boundary, otherwise it will be as zero. Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges. We get the result such as Figure 6:

3 MOTIVATION

Nowadays auto-driving becomes more and more popular. Even 5G take it as one of the main applications. To make auto-driving safer, we want to reduce the latency for getting the relative data. Edge data could be one of the important data sets. To get the edge data, enhancing the computing hardware may be a quick solution. However, optimizing the operation may be more useful and cost less. Therefore, we want to reduce the time for calculating edge data.

4 PROPOSED SOLUTION

4.1 Platform

CuPy: CuPy is an open-source matrix library accelerated with NVIDIA CUDA. It is an implementation of NumPy-compatible multi-dimensional array on CUDA. CuPy consists of cupy.ndarray, the core multi-dimensional array class, and many functions on it. It supports a subset of numpy.ndarray interface. CuPy provides GPU accelerated computing with Python. CuPy uses CUDA-related libraries including cuBLAS, cuDNN, cuRand, cuSolver, cuSPARSE, cuFFT and NCCL to make full use of the GPU architecture [1].

4.2 Parallel Noise Reduction and Gradient Intensity

For speed up these tow phase, the convolution that each pixel has to do independently is the best part to start with. With CuPy library there is a function cupyx.scipy.ndimage.convolve. In this function, the array is zoomed using spline interpolation of the requested order. The difference between 'Noise Reduction' and 'Gradient Intensity' is that Noise Reduction uses a 7 * 7 kernel, and Gradient Intensity uses two 3 * 3 kernels.

4.3 Parallel Non-maximum suppression and Double threshold

In this part, the function in CuPy is not enough to meet our needs. So we need using User-defined custom kernel. This class can be used to define a custom kernel using raw CUDA source. The kernel is compiled at an invocation of the __call__() method, which is cached for each device [2]. Different blocks and grids can have a big impact on kernel performance. In this project, we using 1 dimension grid and 1 dimension block to organize threads. The thread block size is set to 128. The elements in a matrix are stored linearly, and the rows are mainly used for linear storage. The size of shared memory is the same as the data size of all pixels in the thread block. For NMS step, a corresponding thread is declared for each pixel. After setting grid and block, we need to get memory direction for kernel function before entering NMS kernel. Each thread will use the same NMS algorithm as the serial version to determine whether the current processing pixel needs to be suppressed to zero. For Double threshold step, we using two kernel. One is classified kernel, another is hysteresis kernel. Classified kernel is like a simplified version of NMS kernel. It also allocates its own thread to each pixel, and compares the size with the low threshold and high threshold to determine which type of pixel the current processing belongs to. The difference between the hysteresis kernel and the front is that hysteresis kernel only calculates pixels that belong to weak edge.

5 LANGUAGE SELECTION

Python is a major mainstream of today's big data processing. As a widely used language, PyCUDA has good Robustness and an easy-to-use toolkit, including a GPU-based linear algebra library and reduction. Compared with the C language, it has the advantages of easy to use and rapid development. It is easier to use CUDA to convert the program into a parallel version.

Figure 7: Simplex Texture

Figure 8: Complex Texture

6 STATEMENT OF EXPECTED RESULTS

This study will use CUDA to improve the overall efficiency of the system, accelerate it while maintaining the original correctness, and ultimately reduce the expected time to 0.66 times the general performance.

7 EXPERIMENTAL METHODOLOGY

7.1 Method

Our target is to justify our proposed parallel method is better than serial one. We use different type of image with various size as our data set. We try to compare runtime of each tasks separately and show that cuda version is faster than serial version. We also use video to emphasize the huge difference between two method.

7.2 Data Set

We use two sets of data set with different texture complexity. Each data set proportional scaling the picture as 5 images where the size are (1)256*250, (2)512*500, (3)1024*2000, (4)2048*2000, (5)4096*4000.

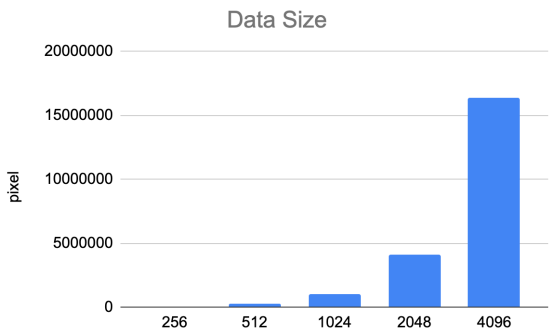

Figure 9: Data Set Size

7.3 Environment

- Operating System
  - ubuntu 18.04
- Hardware Equipment
  - CPU: AMD Ryzen™ 5 2600X
  - Memory: 16 GB
  - GPU: GTX 1060 6G
- Language
  - python 3.6
- Library
  - CuPy
  - Numpy
  - Scipy

8 EXPERIMENTAL RESULTS

We compare runtime and speedup of serial and cuda version in each task. Then, show the speedup and analyze result of each task.

8.1 Gaussian Blur

The runtime in serial and cuda vesrion is proportional to data size. The cuda version is around 9 times faster than serial version in simple texture and around 10 times faster in complex texture. We think the texture complexity would not affect the speedup since gaussian blur only do convolution where the time complexity depends on data size and kernel size.

8.2 Sobel

The runtime in serial and cuda version is proportional to data size. The cuda version is around 4 times faster than serial version in both simple texture and complex texture. We think the texture complexity would not affect the speedup due to the reason in gaussion blur.

8.3 Non-Maximum Suppression

The runtime in serial and cuda version is proportional to data size. The cuda version is around 3500 times faster than serial version. In NMS, it goes through all the points on the gradient matrix to finds the maximum value in the edge directions. The speedup only depends on image size. The speedup in size 256 has bad performance

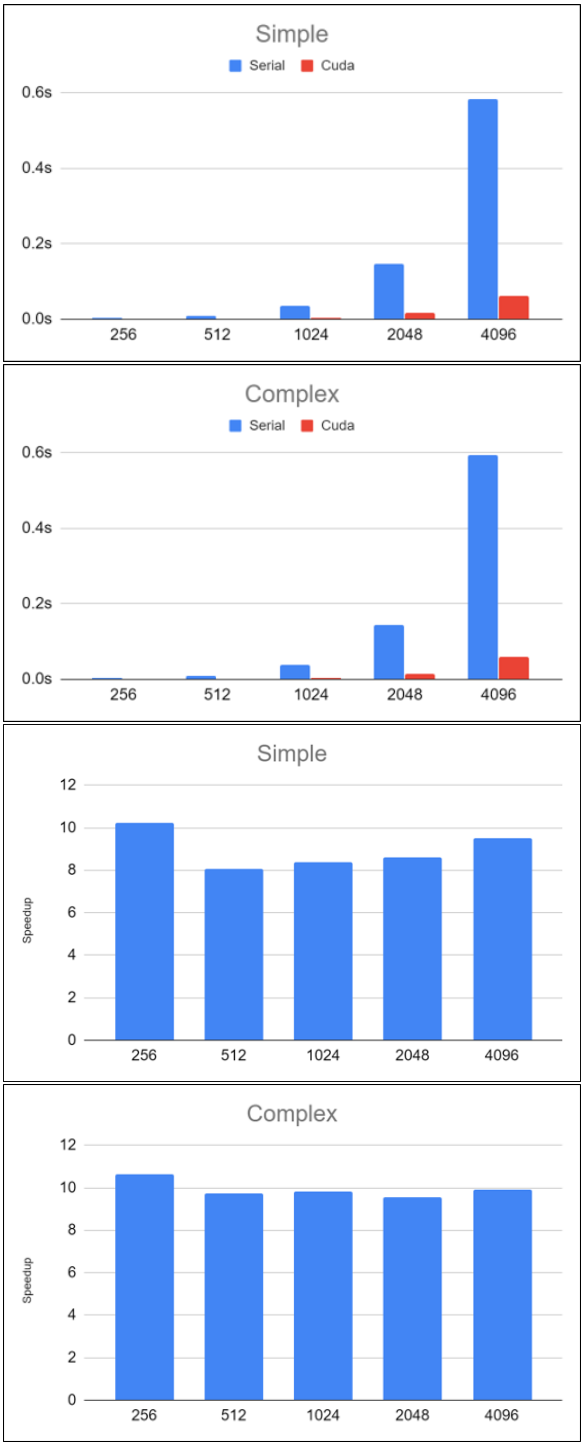

Figure 10: Gaussian Blur Runtime and Speedup

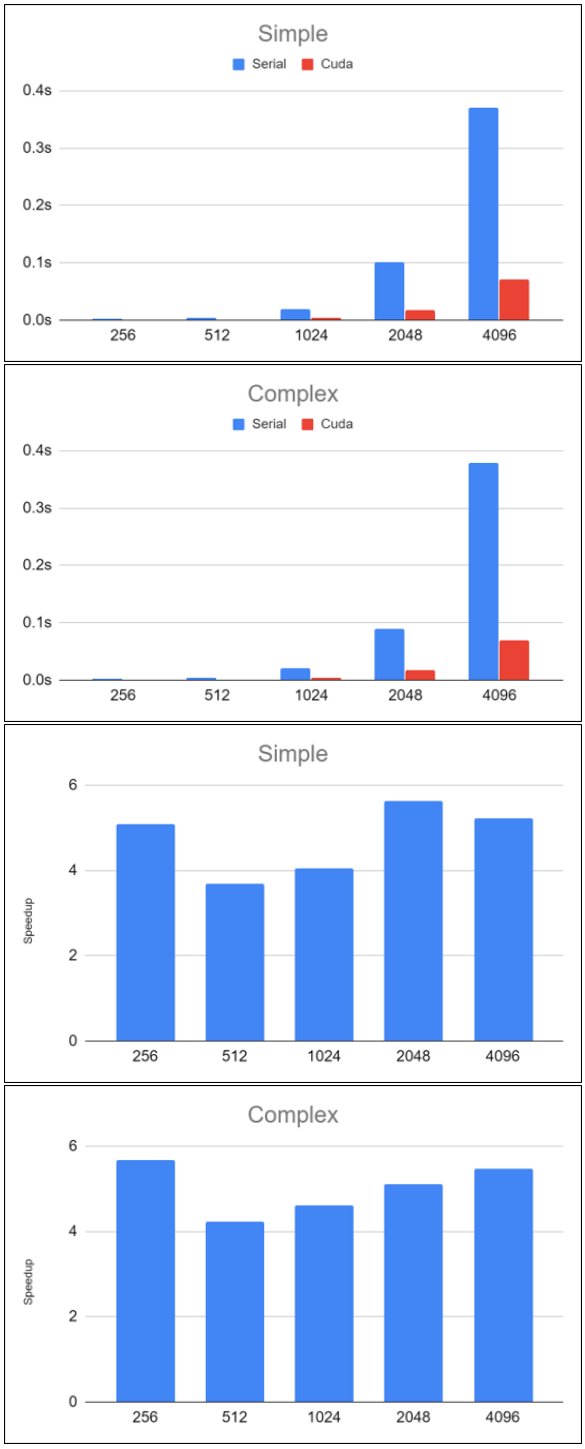

Figure 11: Sobel Runtime and Speedup

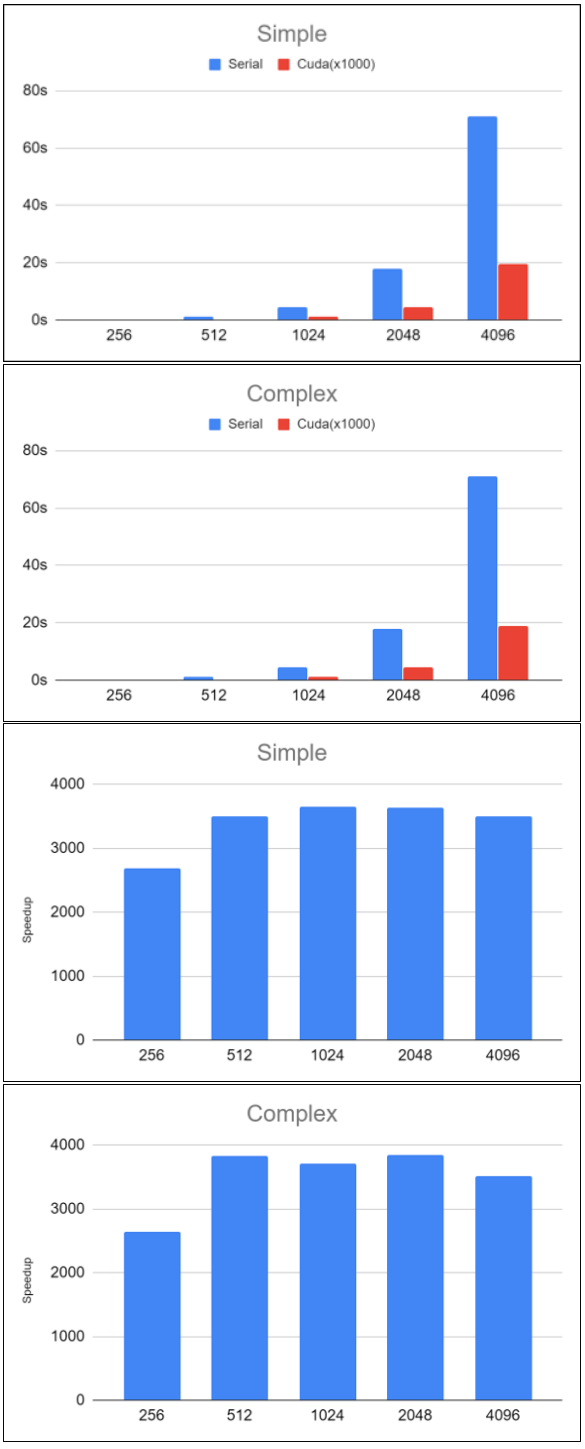

Figure 12: Non-Maximum Suppression Runtime and Speedup

than others. We think that the image size is too small such that the effect of cuda version can not perform as expect.

Table 1: Weak and Strong Pixels

image size	strong pixels	weak pixels
256	1233	3174
512	2151	18980
1024	3731	88939
2048	8801	308968
4096	35933	1241721

8.4 Double Threshold

The runtime in serial and cuda version is almost proportional to data size. The speedup doesn't stable as previous tasks. In double threshold, strong pixels is determined edge and weak pixels can be determined edge if there is strong pixel around. The algorithm only do computation on strong edge and weak edge. The clusters of strong edge and weak edge are not proportional to the size of image. It depends on the texture complexity. We compress same image in different size but the number of clusters may not go proportional. Therefore, we can observe that the curve of time in serial version is not closed to data size in previous tasks. The size 2048 even spends less time than size 1024.

8.5 Complete Canny Algorithm

The runtime in serial and cuda version is proportional to data size. The cuda version is around 400 times faster than serial version. The bad performance on size 256 is due to the small data size the same reason we discuss in NMS.

8.6 Composition of Serial

The runtime of serial version is depends on NMS. The time of gaussion blur and sobel are lower than expectation. We think the reason is that the implementation of convolution we use is not same as we expect. The convolution in SciPy do some subtle tricks. The computation of convolution is transform to frequency domain since multiplication is more efficient (faster) than convolution.

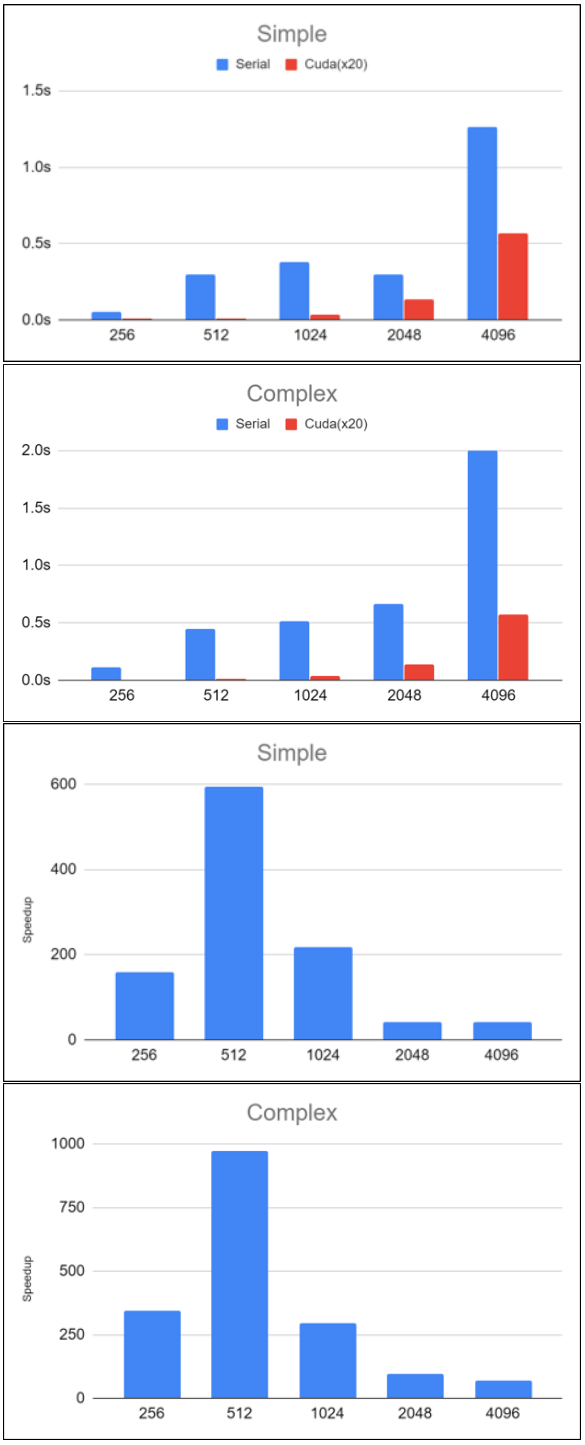

Figure 13: Double Threshold Runtime and Speedup

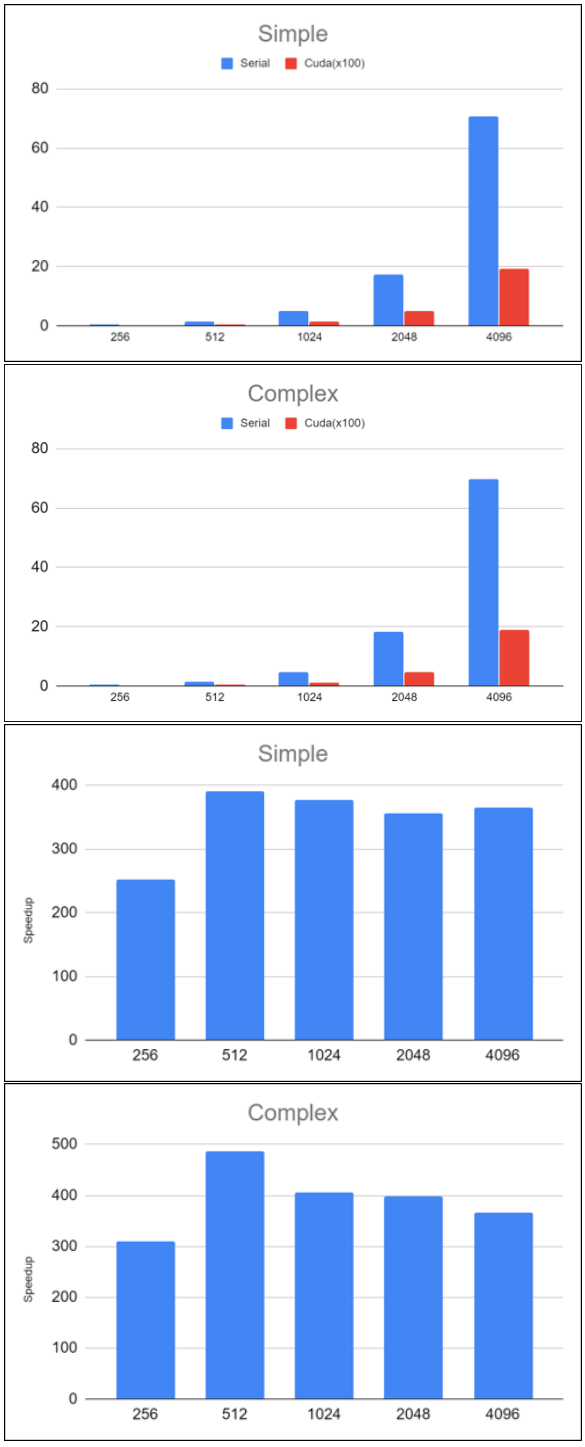

Figure 14: Canny Algorithm Runtime and Speedup

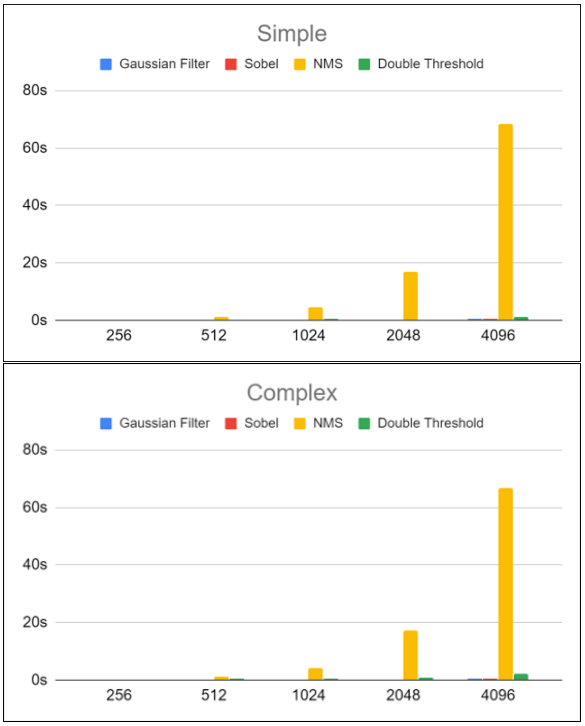

Figure 15: Canny Algorithm Runtime

8.7 Composition of Cuda

In contrast to serial version, each of task occupy parts of time. Since cuda needs to move data from device to host, it spends extra time on data transfer. The convolution in gaussion blur and sobel spends the most of the time. In the same data size, the kernel size of gaussion is less than sobel. But, the time sobel used is more than gaussion. It's because in sobel we do convolution in two direction which means we do convolution twice. It is notable that the NMS dominate the time in serial version. However, the time in cuda version spends less of the time. We successfully break the bottleneck in canny serial version.

8.8 Result

We show the image result of canny algorithm in different image size. We can see that the resolution higher the edge thinner. This is because we can detect the edge more precisely in high resolution and the proportional of edge may get lower.

9 RELATED WORK

Feature detection is the largest and most basic part of computer vision application image preprocessing. Edge Detection is one of the applications. There are two well-known implementations when dealing with Edge Detection problems. The first is to use the Sobel filter[7]. Since the eight points around each point need to be calculated, the Sobel filter takes a lot of time to perform the gradient operation, so many people have tried to parallelize the Sobel filter [6].The second implementation method is Canny filter. Compared to Sobel filter, Canny filter has done more pre-processing to remove

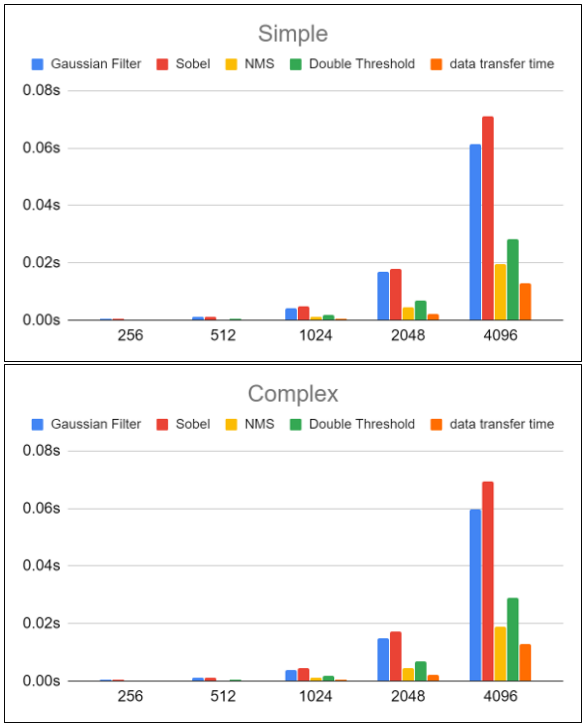

Figure 16: Canny Algorithm Runtime

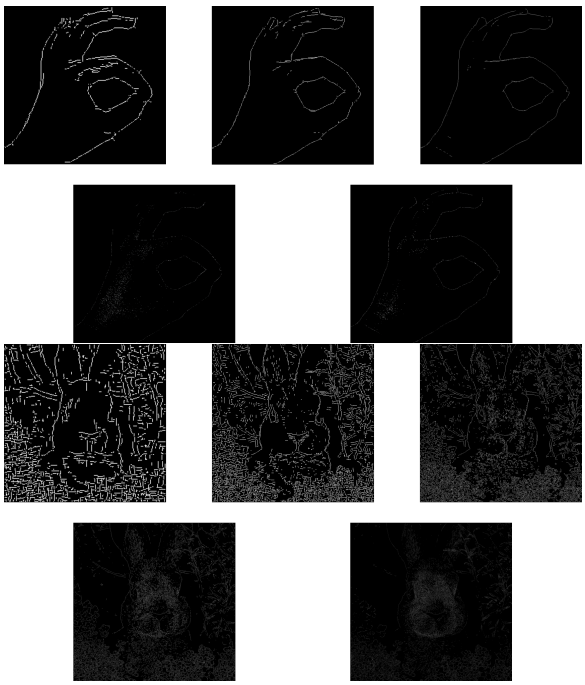

Figure 17: Canny Algorithm Runtime

noise, and uses Non-maximum suppression to find possible edges, making Edge detection better than Sobel filter[4]. In order to reduce the burden on the system, this study focuses on the use of Canny filter, and for the purpose of de-noising, looking for gradient values and directions, and finding possible edges, parallelizing the three steps and performing some related performance comparisons.

10 CONCLUSIONS

In this research, we accelerate canny algorithm by using cuda. Instead of serial processing, we make good use of the property cuda,SIMD, to process data parallely. To make canny faster, we can also improve algorithm in each task besides parallel programming. For instance, the convolution in gaussion blur and sobel can transform to frequency domain to reduce processing time.

Other than performance, the effect of algorithm is the other important issue. The detail of none target edge would be emphasized in high resolution complex picture. For instance, the edge of hand (simple texture) is detected well in high resolution. However, in the complex texture, we want to detect the edge of rabbit but the fur is also been detected. We can use other filter to smooth the picture in different situation to prevent detecting non-target edges. (ex. Bilateral , Guided Filter)

Project Github: https://github.com/james77777778/Parallel-Canny-Edge-Detector

REFERENCES

[1] 2015. *CuPy Overview.* https://cupy.chainer.org/
[2] 2015. *cupy.RawKernel Reference Manual.* https://docs-cupy.chainer.org/en/stable/reference/generated/cupy.RawKernel.html/
[3] Canny edge detector in Wikipedia [n.d.]. . Retrieved December 13, 2019 from https://en.wikipedia.org/wiki/Canny_edge_detector
[4] Juseong Lee, Hoyoung Tang, Jongsun Park 2018. *Energy Efficient Canny Edge Detector for Advanced Mobile Vision Applications.*
[5] Sofiane Sahir. [n.d.]. *Canny Edge Detection Step by Step in Python — Computer Vision.* Retrieved January 25, 2019 from https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123
[6] Sobel filter in Wikipedia [n.d.]. . Retrieved Oct 24, 2019 from https://en.wikipedia.org/wiki/Sobel_operator
[7] Soonjong Jin, Wonki Kim, Jechang Jeong 2008. *Fine directional de-interlacing algorithm using modified Sobel operation.*