Junye (Hans) Chen, Zhan (Patrick) Dong
15-418 Fall 2017
December 10, 2017

Final Project Report

Parallel License Plate Recognition

Summary

In this project, we implement parallel license plate recognition using CUDA on the GPU of GHC machines. The program takes in a PNG file as input, and outputs the plate region with bounding boxes identifying the characters on the plate. We compare the parallel solution with the sequential solution on CPU and observe a good speedup. It also has a satisfying accuracy.

Background

Program Overview

In this project, we implement our program from scratch based on the algorithm from the paper written by Mahesh Babu K and M V Raghunadh[1]. We implement a multi-step algorithm involving many classical image processing algorithms, and use our own ideas to do the parallelization for each step. The general process of our program is shown in the flow chart below:

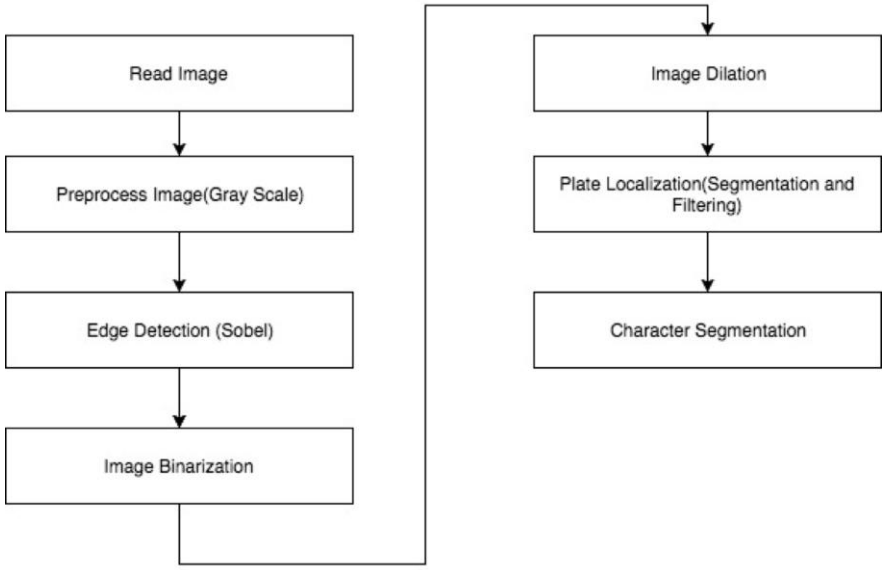

Figure 1. Algorithm Overview

The input of our program is a PNG file that contains a photo of the back of a car with a license plate, and the ideal output is a PGM file (grayscale of the original image) that contains the plate, with bounding boxes around character segments. Throughout the process, we keep making changes on the array representing the image, trying to extracting the important regions and getting the information we want.

Computation, Locality and Dependencies Analysis

The entire algorithm is, as shown above, quite computation intensive. Basic operations like applying a kernel to every pixel in the image is extremely suitable for CUDA, since we execute simple calculations for a large amount of times, and GPU is really good at such operations. Although these simple operations don't require that much time by default, there are some relatively complicated operations like binarization and image segmentation that provide us with a lot of potentials for speedup. These parts will be the major source of our speedup.

The locality of most parts of this program is good, since we are only looking at neighboring elements in the array most of the time. However, we use a union find data structure to represent

connected components in the image during segmentation. This requires a lot of pointer chasing, and we have to access the array in a chaotic order. That may result in very bad locality, with lots of cache misses.

The most parts of the program also have good dependencies requirements. In many parts of the program, every pixel can be processed at the same time. But in the segmentation part, the result of each pixel depends on the three pixels on its top and the pixel on its left, so we need to find a wise way to parallelize the segmentation step.

From the analysis above, we can observe that the segmentation part is the most challenging part in our program, with bad localities and strong dependencies. It is also the most computation intensive part, so it also provides a lot of potentials for speedup. A good parallelization of this step will result in a very satisfying general speedup.

Approach

General Information

For this project, we use the programming language C and C++. We also use built-in Linux libpng for reading and processing PNG files, and CUDA to implement parallelization.

We run all our experiments on GHC machines with GPU Nvidia GeForce GTX 1080. For the sequential part of the code, we have Intel(R) Xeon(R) CPU E5-1660 v4.

We write most parts of our codes from scratch, and implement most of the image processing operations on our own, so that we can parallelize without any constraints.

Since our project consists of different image-processing steps, we believe that CUDA is the most suitable way for parallelization. We mainly use CUDA in two ways. For some of the steps, we

generate a lot of threads, and deal with the operations of 1 pixel on 1 thread. For the rest of the steps, we break the entire graph into blocks, and generate corresponding number of threads, where each thread takes care of a block. We use arrays with size of the number of threads to get partial results from each block, and combine the subset sums and results, obtaining our final results.

Our entire program consists of several steps, and we parallelize each step to ensure maximum overall speedup.

Preprocess Image

After taking a PNG image as input, we apply our grayscale ingredient on every pixel of the image, and make it into an array of type unsigned char. The ingredient we use for this program is:

$$GreyValue = 0.144 \cdot RedValue + 0.587 \cdot GreenValue + 0.299 \cdot BlueValue$$

The result of grey-scaling is as follows:

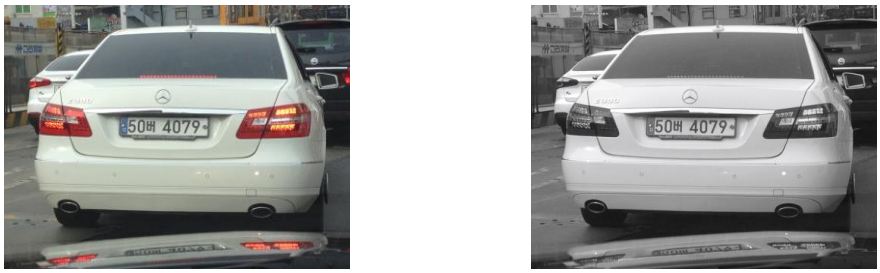

Figure 2. Result of Preprocessing

We obtained our sequential code for preprocessing from the sample code of libpng[2]. We parallelize preprocessing using CUDA, by assigning each pixel to a CUDA thread. We copy the information from PNG file to the global memory of the device. In the kernel, we apply the ingredient to the current RGB values.

Edge Detection

In this program, we use Sobel operator to do edge detection. The method involves applying two 3 x 3 kernel that are convolved with the original image. The idea behind Sobel edge detection is to use two kernels (shown below) to detect the sharp horizontal and vertical gradient.

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Kernel to detect horizontal sharp gradient Kernel to detect vertical sharp gradient

Once we get the two results (call them g_x and g_y) after applying two kernels at one pixel, we compute $\sqrt{(g_x)^2 + (g_y)^2}$ and assign it to the current pixel. This process helps to highlight the sharp gradient in an image, and specifically help to highlight the position of the license plate in our project, because there is usually strong contrast between the license plate and nearby regions. This step may seem trivial at the beginning, but in order to obtain good result, we need to normalize the image after convolution, which involves finding the maximum and minimum pixel value in the whole image. Therefore, we need to use two CUDA kernels to achieve this, where the first one is to find the maximum and minimum pixel values after convolution, and the second one is to apply normalization to each pixel value. The normalization equation looks like

$$255 * \frac{(convolution\ result - min\ pixel\ value)}{(\max pixel\ value - \min pixel\ value)}$$

Since CUDA divides the image into blocks, and communication between blocks is rather expensive, we use shared maximum and minimum variables within each block. Each block finds its local minimum and maximum pixel values, and then the first pixel within each block updates the global maximum and minimum pixel values among the whole image. All the updates are

achieved through CUDA atomic operations, so no updates will be lost. We observe that the Sobel operator works extremely well for greyscale images.

Figure 3. Result of Sobel Edge Detection

Image Binarization

To get the image prepared for later image processing steps such as dilation and segmentation, our next step is to binarize the image, making each pixel to have either 0 (black) or 255 (white) as its value. At first we decided to use a static threshold 127 to binarize the image. This is quite intuitive, but not that satisfying when we try our algorithm on various graphs. Since the input image can have different light conditions and color themes, using a static threshold sometimes result in an uneven classification of pixels. Instead, we need to use a dynamic way to determine the threshold for binarization, so that the characteristics of the image can be fully extracted. The algorithm of acquiring the threshold is as follows[3]:

```
while (new threshold != old threshold) {
        above sum = sum up all pixels above threshold
        above count = count # pixels above threshold
        below sum = sum up all pixels below threshold
        below count = count # pixels below threshold
        old threshold = new threshold
        new threshold = above sum / above count + below sum /below count
}
```

The idea is keep updating the threshold, thus finding the optimal threshold that evenly break the pixels into the white ones and black ones.

We built several kernels in CUDA to parallelize this step. The computation in this step is calculating the sum of all pixels below/above the threshold and count the number of pixels below/above the threshold, so we build kernels to do those calculations in parallel.

The general idea for these kernels is the same: we break the entire image into blocks. To ensure good temporal locality, we think of the entire image as a 1D array, and break those blocks following the idea. We also have arrays of size of the number of blocks. Each cell in the arrays records the partial results. We then combine the partial results in the host function. Since each thread only updates one spot in the array, and no element in the array is accessed by multiple threads at the same time, there will be no race conditions and reader-writer problems. This commonly-used partial result parallelism works extremely well under such settings.

Image Dilation

To turn the plate region into a single connected component, our next step is to apply dilation on the edge graph. The basic idea of dilation is very simple. By applying an x radius and a y radius, each pixel will check its neighbors in the $(2x + 1) \times (2y + 1)$ matrix that centers at the pixel. If any of the neighbors is a foreground pixel, the pixel itself will also become a foreground pixel. The effect of dilation is shown with the example below:

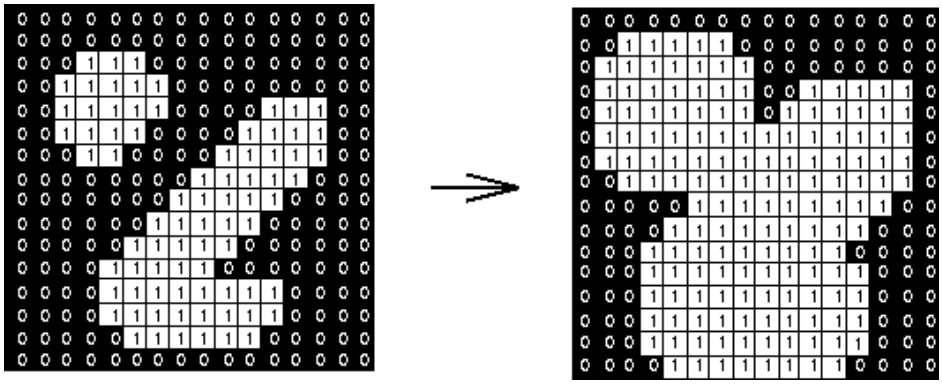

Figure 4. Dilation: a 3×3 square structuring element[4] (www.cs.princeton.edu/~pshilane/class/mosaic/).

The structuring element we use in this program is based on the width and height of the image, but also have a lower bound. a structure element of 4 x 8 works for many of the images.

Figure 5. Dilation on the Edge Graph

After the dilation, we can observe that the edges in the graph swallow, and form into larger connected components. Among them, the plate region is very unique, since it has a rectangle shape, and is isolated from other connected components. This provides the features we need for segmentation and identification of regions.

We again use CUDA to parallelize this step. we create a new array on device to save the result image. We cannot directly change the original image, since the results of each pixel have dependencies on the old values of the graph. Our graph will be different given the order of pixels we process if we directly change the values. Then we run CUDA with a large amount of threads, where each thread takes care of a pixel. In the thread, we look at the neighbor elements to see if there is a foreground pixel. If we find one, we change the value in the new image and terminate the

thread. Finally, we copy the new image back. The parallelism for dilation is not optimal, since it involves memory copy and other time-consuming operations. However, we have to do so to ensure the correctness of the step.

Segmentation

Segmentation is the most important step in our entire algorithm. The basic idea is to the find the connected components of the foreground pixels in the image. We use segmentation to achieve two goals: one is to find the specific regions that contain the license plate, and the other is to identify the character segments.

The first algorithm we propose is sweeping line algorithm. The algorithm is quite simple: by using a sweeping line to go from left to right, top to bottom, we find the bounds for the foreground pixel components in the image. However, this algorithm only works for simple images, where different components do not appear in the same row or column of the image. To find connected components on a more complicated image, we apply a new two-pass algorithm from another paper[5] using the idea of union find.

We go through the entire graph for two passes. In the first pass, we loop through the pixels row by row, from left to right. When we are at one pixel, we check its left pixel, and the three pixels at the top, because these four pixels have already been labelled. If any of them is a foreground pixel, and already labelled, the current pixel inherits its label. However, it is possible that multiple neighbors already have labels. In this case, we need to find the canonical representative label of each neighboring pixel, and assign the current pixel the smallest label among the canonical representatives. We use union find data structure to maintain the canonical representatives of labels. In the case where multiple neighbors have labels, we also need to update union find data

structure if one of the neighbors doesn't have the smallest canonical representative. The figure below shows a situation where multiple neighbors of the current pixel have already been labelled. The pixel at row 3 column 4 has left neighbor with label 1, and upper neighbor with label 2. Therefore, we assign the pixel with label 1, and update the union find data structure that points label 2 to 1.

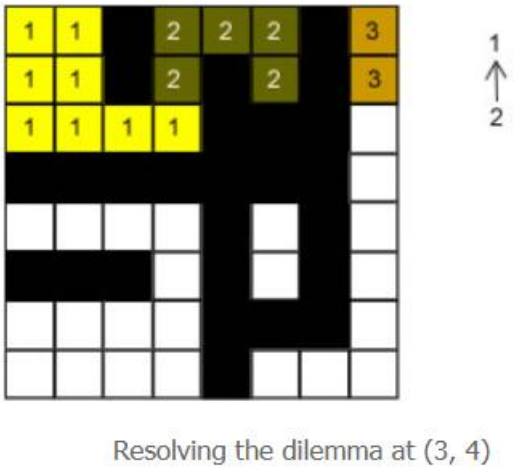

Figure 6. Example of Label Conflict Resolution in Connected Component Analysis[6]

In the second pass, we look into our union find data structure, and update the labels. In the figure above, we will update all the pixels with label 2 to label 1, so that the pixels with labels 1 and 2 are connected into one component.

This algorithm turns out to be incredibly slow on large images. In order to speed it up, we use CUDA to parallelize the algorithm. However, the algorithm is inherently sequential because it has a lot of dependencies. Every pixel has to wait for its left and upper pixels to label themselves first. After some research, we discover some paper[7] that does analysis on parallelizing this algorithm. We propose that we can reduce the problem size by applying blocking on the original image. Essentially, we can apply the two-pass algorithm on each block of the image first. Then we need to resolve all the label conflicts on the edges, and update labels again. This idea fits perfectly into the CUDA structure. We divide the image into blocks of 32x32, and then let the first thread within

each block run the two-pass algorithm. After the first thread is done, all the threads update its pixel label by looking into the union find data structure. In order to reduce communication overhead, we take advantage of CUDA shared memory again, and place the union find data structure inside shared memory, so all the threads within the same block can update their labels in the second pass by accessing the shared memory. An important technique to reduce the complication of label conflict resolution in the next step is that we always start a different label count for each block. For example, assume block size is 32x32. If block 1 starts assigning label from 1, block 2 should start assigning label from 32x32 + 1. This prevents having the same label in two different blocks.

Now we need to resolve any label conflicts on the edges between blocks. Initially, we think about parallelizing this step, but along the process of label conflict resolution, we need to read and write into a global union find data structure, which involves synchronization overhead. We decide to stick with a sequential version of this step first, and the timing of this step turns out to be surprisingly fast. We think this is because there are not that many pixels to look at on the edges, so parallelizing this step doesn't provide a lot of speedup on our solution.

Last step involves updating label at each pixel, which is trivially parallelizable because every pixel just has to look into the union find data structure and find its canonical representative. Every pixel can do it at the same time, which provides a lot of parallelization potential.

Figure 7. Finding Plate Region from the Original Image

Filtering

After finding all the connected components in the image, we need to filter out the unnecessary components, and only save those potential components that contain a license plate.

We apply filtering twice. We first use it to find the plate region among all the connected components in the original image. We notice that the plate is normally a rectangle region, where the height of the rectangle a lot smaller than the width of the rectangle. Besides, since it is a relatively small part of the car, and its size is also bounded. We use the width-height ratio constraint to eliminate regions that have similar size, but of a different shape. We use the size constraint to eliminate regions that have a reasonable shape, but is either too small (noises in the image) or too big.

Then we apply filtering to find character segments on the plate. We also have size and width-height ratio constraint, so that we can get rid of the confusing features on the plate, and only focus on the actual character segments.

We spend a long time tuning the parameters for both filtering steps, so that plates from most of the images fit into the region, and any unqualified regions can be eliminated. This is quite necessary for the general speed of the algorithm, since whenever we detect a potential region, we use our segmentation technique to find character segments. If we have to do character segmentation on all the connected components, it will be a huge waste of time.

Figure 8. Successful Detection of All Segments in the Original Image

Results

We evaluate our solution in accuracy and speed.

Accuracy

We define the accuracy of our project to be the percentage of test images where we can successfully extract all character segments. The errors can be mainly classified into two types:

1. Cannot find the plate region. This often occurs when the plate and the nearby edges form a huge connected component.

2. Found the plate region, but the binarization of the plate breaks the image. This often occurs when the background of the plate and the color of the numbers don't differ a lot. As a result, they will be classified to either both black or both white. Another result for incorrect binarization is the wrong number of number segments. This often occurs when the lighting condition is not optimal. As a result, the number is connected to the edge of the plate, and form a huge connected component, which we can't identify as a valid character segment.

We test our solution on a total of ten images, and the results are listed in the following table:

Absolutely Correct	7
Successful Detection of Plate Region, but Incorrect Number of Segments Identified	2
Failed to Detect Plate Region	1

Table 1. Accuracy Report

Here we show three examples of successful detection. The first example is shown with a detailed procedure of how everything is connected.

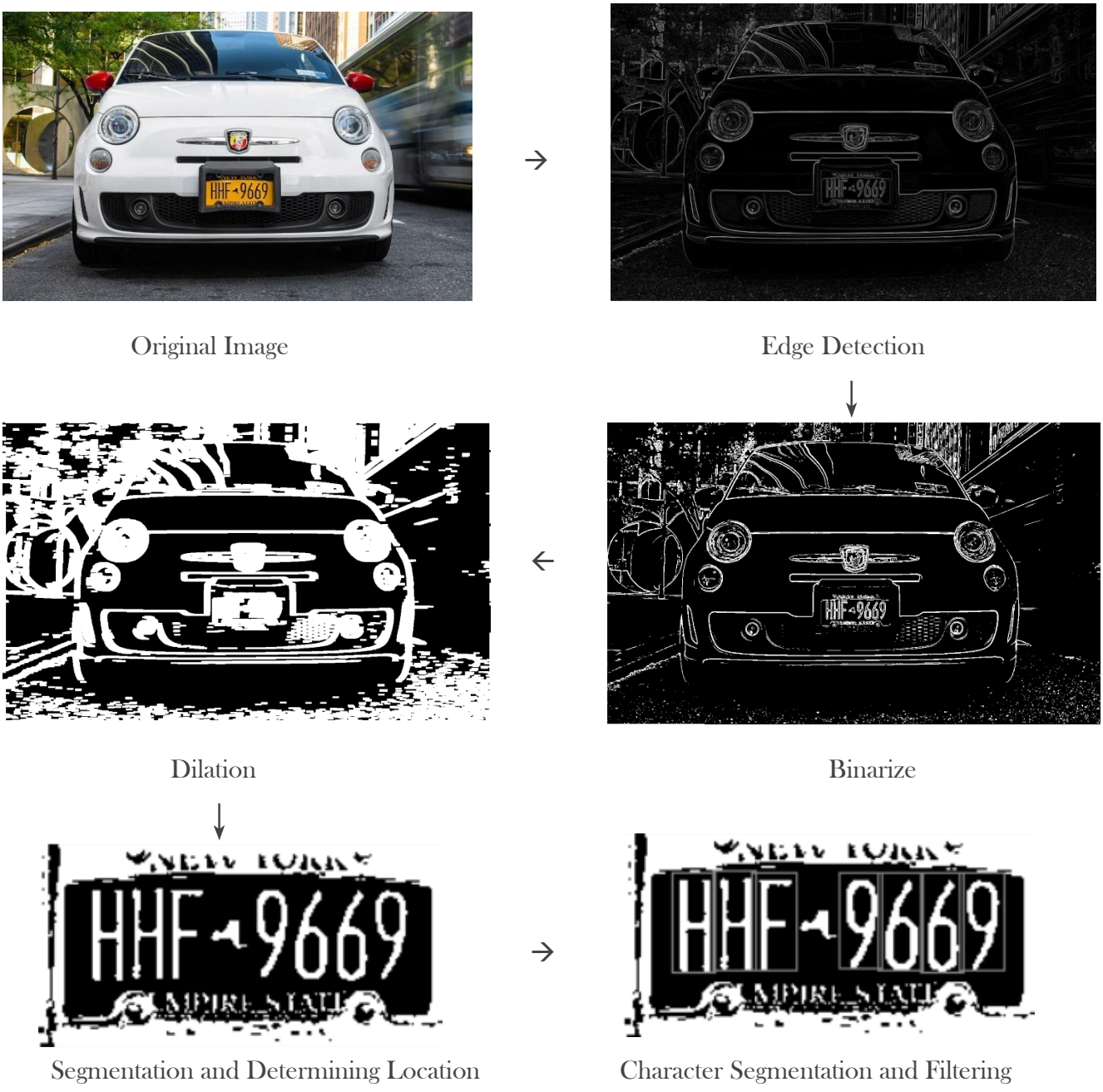

Figure 9. Successful Detection Example 1

Successful detection example 2:

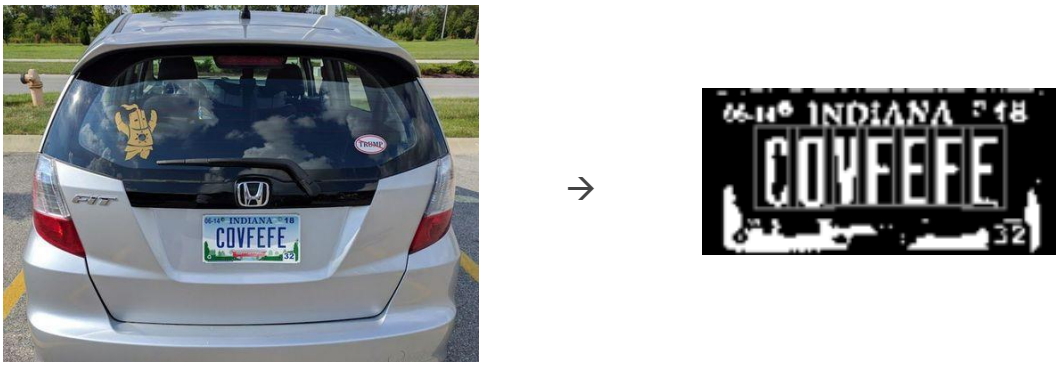

Figure 10. Successful Detection Example 2

Successful detection example 3:

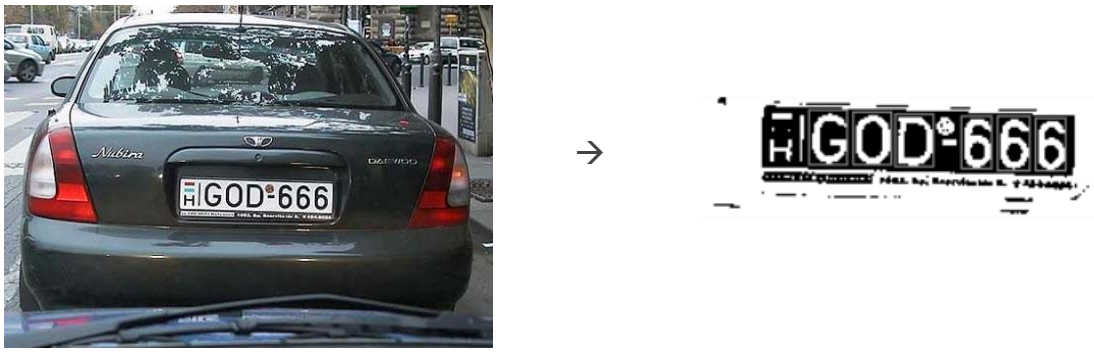

Figure 11. Successful Detection Example 3

An example of failure to recognize character segments is as follows. We can see the original image doesn't have good contrast between the character segments and the plate. After we locate the plate and binarize the region, the characters are not very obvious, so our segmentation algorithm cannot detect connected components.

Figure 12. Incorrect Character Segment Detection

An example of failure to recognize plate region because it is connected to other parts of the car after dilation:

Figure 13. Incorrect Recognition of Plate Region

Speed

We observe significant speedup of our CUDA implementation against our sequential CPU implementation. We use built-in clock function for all the timing data, and here are the data on three different input image sizes:

Input Image Size: 534x401

Steps	Sequential Time (ms)	Parallel Time (ms)
Preprocess PNG File	10	<1
Edge Detection	20	<1
Binarizing	10	<1
Dilation	10	<1
Plate Location and Character Segmentation	1350	70
Total Time	1410	70

Table 2. Runtime on Input Image 1

Input Image Size: 1024x780

Steps	Sequential Time (ms)	Parallel Time (ms)
Preprocess PNG file	40	<1
Edge Detection	70	<1
Binarizing	20	10
Dilation	40	<1
Plate Location and Character Segmentation	10460	100
Total Time	10630	110

Table 3. Runtime on Input Image 2

Input Image Size: 1452x1205

Steps	Sequential Time (ms)	Parallel Time (ms)
Preprocess PNG File	90	10
Edge Detection	120	<1
Binarizing	20	20
Dilation	130	10
Plate Location and Character Segmentation	35410	100
Total Time	35770	140

Table 4. Runtime on Input Image 3

The following is the sequential runtime vs problem size. We define problem size as the number of pixels in the image. We can see the runtime significantly increases as problem size grows. We can observe that when problem size increases from 500000 to 1000000, the runtime increases by almost 10 times! A quadratic equation probably fits the data well, which means it is going to be take significantly longer time to run the sequential implementation on larger input images.

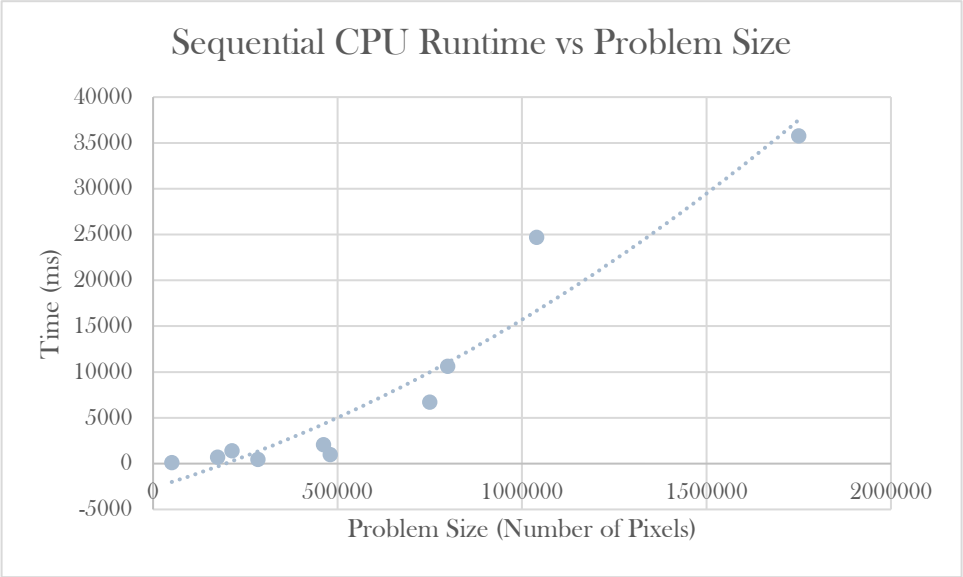

Figure 14. CPU Runtime vs Problem Size

The following is the total CUDA GPU implementation runtime vs problem size. We can see the runtime increases approximately linearly with the problem size. When problem size is 500000, runtime is approximately 50ms, and it increases to approximately 100ms when problem size is doubled to 100000. The linear scale makes sense because our parallel solution essentially reduces a $O(n^2)$ complexity down to O(n) (n is defined as width or height), because we apply blocking on the image. The only non-parallel step is edge resolution, where we only check the edges that have label conflicts, but the runtime of that is scaled approximately linearly with input size.

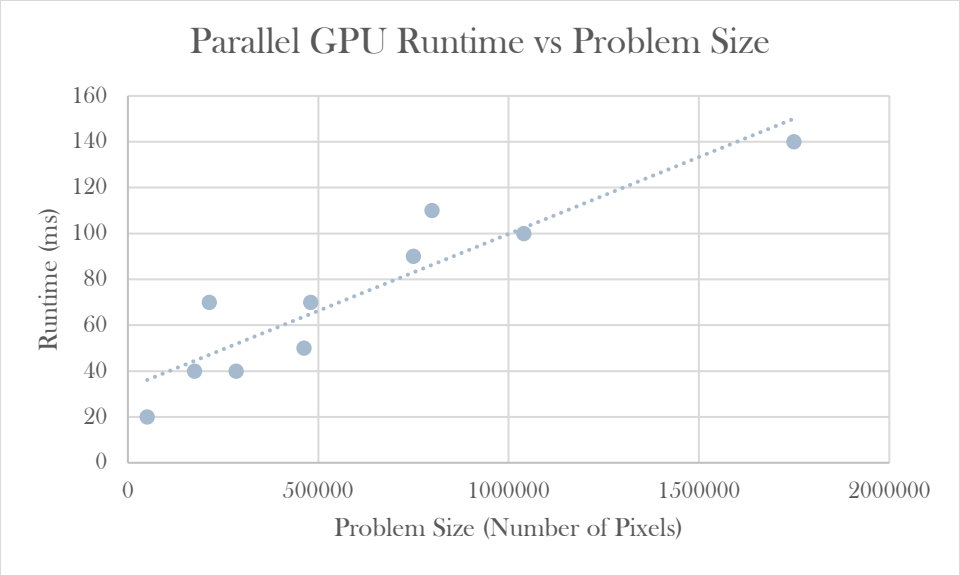

Figure 15. GPU Runtime vs Problem Size

The last graph is the relative speedup of our CUDA implementation vs CPU implementation. We can see as problem size increases, the speedup significantly increases. This further confirms that our parallel solution successfully reduces the problem size. With input size 1452x1205, our solution achieves 255x speedup vs the CPU implementation, where we can clearly see the potentials of the parallelization techniques.

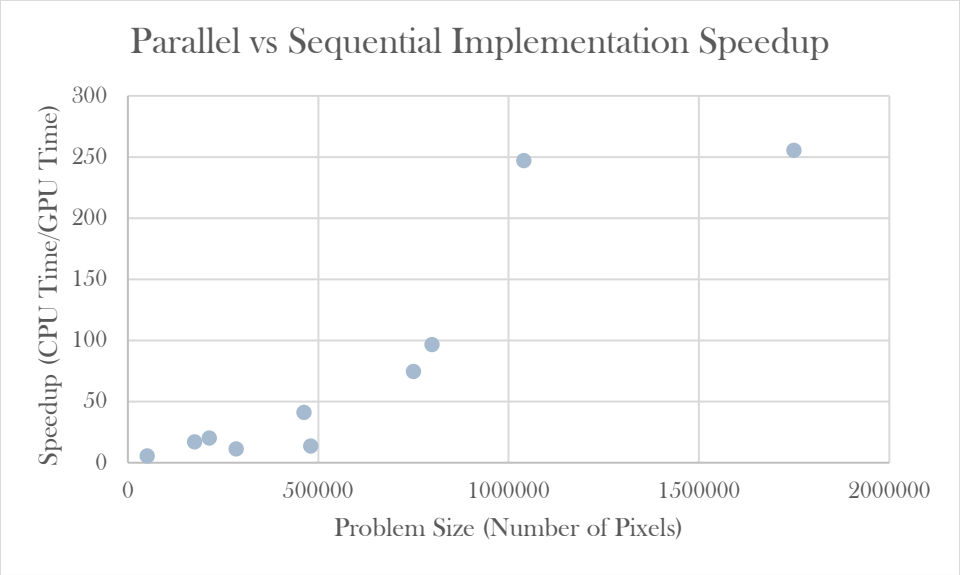

Figure 16. Parallelization Speedup

Limitations

Although we observe a satisfying speedup, there are still many aspects that limit our speedup. In the timing part, we can observe that some parts like preprocessing don't have an obvious speedup. This results from the fact the parallelization version of these steps have large memory copy operations. We have to copy our result from host to device, or copy the new image onto the old image. Since the preprocess computation on each pixel is not that time-consuming by default, the overhead of copying these elements from memory is significant. As a result, we don't see a good speedup for these parts of the program.

We also have a decent accuracy, but the accuracy can be better if we apply more image processing steps or tune the parameters more. We don't have chance to try some potential improvement techniques due to time limitation, but it is a very good start for further research.

Work Distribution

We both spent more than 60 hours on this project. I (zhand) implemented the sequential and parallelized version of preprocessing, dilation, filtering and the parallelization of binarization. Besides, I was also in charge of tuning the parameters, and writing the major parts of our essays. Hans (junyec) was in charge of the sequential and parallelization of Sobel operator and segmentation, and the sequential version of the binarization. Besides, he also did the speed tests, and wrote the rest parts of the essays. We tried to work on this project in a parallel way: everyday we focus on two independent tasks and try to solve it on our own, and merge our results into a complete project. The work is distributed evenly among us on a 50%-50% basis, But since Hans

(junyec) was in charge of the parallelization of segmentation, the most challenging part of our program, he also spent longer time on coding and debugging.

References

1. Babu, K Mahesh, and M V Raghunadh. "Vehicle number plate detection and recognition using bounding box method." 2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), 2016, doi:10.1109/icaccct.2016.7831610.

2. A simple libpng example program, zarb.org/~gc/html/libpng.html.

3. Sridevi, Thota, et al. "Morphology Based Number Plate Localization for Vehicular Surveillance System." *International Journal of Modern Engineering Research (IJMER)*, Mar. 2012, pp. 334–337., www.ijmer.com/papers/vol2_issue2/BC22334337.pdf.

4. Morphological Image Processing, www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm.

5. Sinha, Utkarsh. "Connected Component Labelling." *AI Shack - Tutorials for OpenCV, computer vision, deep learning, image processing, neural networks and artificial intelligence.*, aishack.in/tutorials/connected-component-labelling/.

6. Sinha, Utkarsh. "Labelling connected components - Example." *AI Shack - Tutorials for OpenCV, computer vision, deep learning, image processing, neural networks and artificial intelligence.*, aishack.in/tutorials/labelling-connected-components-example/.

7. Fei, Yang. "Yang-Fei-F2010.Pdf." *Index of /Faculty/Miller/Courses/CSE633*, www.cse.buffalo.edu/faculty/miller/Courses/CSE633/.

Other Paper we got inspiration from

Pardeshi, Chetan, and Priti Rege. "Morphology Based Approach for Number Plate Extraction." *Proceedings of the International Conference on Data Engineering and Communication Technology Advances in Intelligent Systems and Computing*, 2016, pp. 11–18., doi:10.1007/978-981-10-1678-3_2.

Zhai, Xiaojun, et al. "License plate localisation based on morphological operations." *2010 11th International Conference on Control Automation Robotics & Vision*, 2010, doi:10.1109/icarcv.2010.5707933.