Parallel Sobel Edge Detection Algorithm using OpenMP and CUDA

Guanyue Bian (guanyue_bian@berkeley.edu)
Jingjing Wei (jingjingwei@berkeley.edu)
Yaowei Ma (yaowei_ma@berkeley.edu)
Github Link: https://github.com/YW-Ma/CS267-Final-Edge-Detection

1. Introduction

Edge detection is a critical task in computer vision that aims at identifying edges or curves in digital images mainly based on the value of pixels changing sharply and can be further used in image classification and object detection. A famous edge detection algorithm named the Sobel algorithm was designed to use a 3*3 kernel to perform convolution on the images and output values based on surrounding neighbor pixels.[1] However, when the image size is large, it would take a long time to process the image using the Sobel operator. So we plan on implementing the serial version Sobel algorithm and then parallel it with OpenMP and CUDA to speed up the algorithm efficiency and analyze the results of paralleled algorithms.

2. Serial Edge Detection Algorithm

2.1 Workflow

As shown in Figure 1, the workflow of our algorithm contains the following steps. First, we input a PNG image and use the third-party library LodePNG[2] to decode the PNG image into an RGBA image with four channels. After that, we convert the RGBA image into a grayscale image with only one channel. Then we use the Sobel operator for edge detection. The output of the Sobel algorithm is a matrix containing edge information. The width and height of this matrix are equal to the width and height of the input image. After that, we do a grayscale stretch on the edge matrix to obtain an edge grayscale image whose value from each pixel conforms to the standard from 0 to 255. After that, we convert the grayscale image to an RGBA image with four channels. Then we use LodePNG to encode the RGBA image into a PNG image as the output.

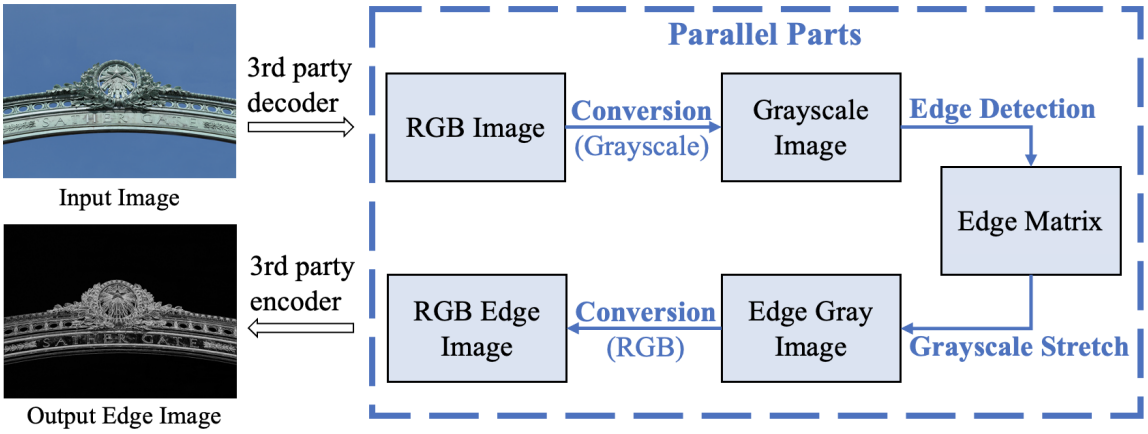


Figure 1 workflow of Sobel Edge Detection Algorithm

2.2 Data Structure

In this project, we customized the data structure to complete the task of edge detection. First, we define the variables named width and height with the type of uint32_t. Because the edge detection task does not change the width and height of the input image and output image, here we only define width and height once, they will be used by input RGBA image, input grayscale image, output grayscale image, and output RGBA image. After that, we define a 1D array named input_image_data with the type of unsigned char used to store the pixel value of the input RGBA image. The size of input_image_data is width * height * 4 because the RGBA image has four channels, and each pixel has four values from 0 to 255. Next, we define a 1D array named output_image_data with the type of unsigned char and size of width * height * 4, which is used to store the value of the output edge RGBA image. At the same time, we also define a 1D array named gray_input_image_data and a 1D array named gray_output_image_data with the type of unsigned char. The size of these two variables is width * height because the grayscale image has only one channel.

To implement Sobel operator edge detection, we define two 3*3 2D arrays, with the type of unsigned char named sobel_kernel_x and sobel_kernel_y separately. These two variables are convolved in the x and y directions, respectively. In addition, we define grad_x_mat_data and grad_y_mat_data with the type of int16_t and size of width * height, which are used to store the results of convolution of the image in the x and y directions, respectively.

2.3 Function Implementation

The serial Sobel edge detection algorithm designed by us mainly comprises the following functions. RGBA_to_grayScale() function converts an RGBA image to a grayscale image. The inputs of this function are width, height, input_image_data and gray_input_image_data. In this function, we use a for loop to traverse all pixels' values, add the values of the three channels of R, G, and B, and divide them by three as the grayscale value.

The second function is cov3(). The input of this function is width, height, gray_input_image_data, sobel_kernel_x/sobel_kernel_y, grad_x_mat_data/grad_y_mat_data. This function implements the Sobel operator. In this function, since the convolution output will miss the values of the first column, the last column, the first row, and the last row, we pad the missing values. We first padding the first row and the last row, the value of the first row is copied from the second row, and the value of the last row is copied from the penultimate row. Then pad the leftmost column and the rightmost column with the value of the adjacent column. We call this function twice to do edge extraction in the x and y directions.

The inputs to the post_processing() method are width, height, grad_x_mat_data, grad_y_mat_data and gray_output_image_data. In this function, first, we create a temporary variable called temp_mat with the type of int16_t and size of width * height. We use a for loop to traverse all the values of grad_x_mat_data and grad_y_mat_data and assign values to temp_mat based on grad_x_mat_data and grad_y_mat_data . Then we use a for loop to iterate over all the values in temp_mat and find the largest value. After that, we do grayscale stretching. We use a for loop to

traverse all the values in temp_mat, set the maximum value of temp_mat to 255, and scale other values proportionally. The result of grayscale stretching is stored in gray_output_image_data.

The last function in our serial algorithm is grayScale_to_RGBA(). The inputs of this function are width, height, gray_output_image_data and output_image_data. In this function, we use a for loop to traverse the values of all pixels, assign the values of the R, G, and B channels as grayscale values, respectively, and set the value of the A channel to 255 (fully opaque).

3. Parallel Edge Detection Algorithm

3.1 OpenMP

Once we have figured out how to implement the serial version of the Sobel operator algorithm, we find it becomes a problem when the picture size is large. So for the next step, we would like to parallelize the edge detection process. According to the Sobel operator algorithm workflow, we first try to use OpenMP to speed the algorithm up.

Since OpenMP uses shared memory, and all different threads would like to access the same memory space if they want to read or write the same variable, the main cost of more processing is waiting for race conditions. Based on the workflow, we decided to parallel four parts which are conversion from RGBA images to grayscale images, edge detection, grayscale stretch, as well as conversion from grayscale to RGBA images. When we want to convert RGBA images to grayscale, we take the mean value of R, G, B, and A channels for each pixel. Since each pixel is independent, so there will not be any race condition between different threads, we parallelize the for loop of conversion. At first, we tried to manually divide the whole image into threads and make different threads take charge of a certain amount of pixels using the thread id of the index. Later, we discovered that OpenMP builds in function #omp parallel for could achieve faster performance. We assume this is because it could utilize cache line data more than we manually parallel for loops.

When parallelizing the edge detection process, since it is a convolution kernel moving around the whole image, and it reads from the original image and stores the convoluted results in a new data structure, we could use parallel for again. Also, after the parallel segment, we need to include an *omp barrier* to ensure all threads have finished the edge detection process before moving to the next step. But because the *parallel for* has an implicit barrier by the end, we do not need to emphasize that again in our code. For the grayscale stretch part, we want to normalize the pixel value and stretch it to 255 to convert it back to an RGBA image in the future. For normalization, we need to find the maximum value within the whole image. We could only parallelize the divide part, but we could not parallelize the finding maximum value part since in order to find the largest one, we need to compare each of them.

At last, for converting grayscale images back to RGBA images, we also use parallel for to make the for loop automatically divided and assigned to different threads, which is the same technique we use for the first part.

3.2 CUDA

In addition to OpenMP, we also tried to use CUDA for parallelizing the edge detection algorithm. CUDA has a different memory architecture than OpenMP. Each CUDA block has its local memory. It will be time-consuming if data is transferred between GPU memory and CPU memory unnecessarily frequently. Same as OpenMP, we also focus on parallelizing the four parts of the whole workflow.

We design a data structure named pixel containing components including values of R, G, B, and A values, and then we use this as the structure of each pixel for the following process. We construct the pixel structure at the beginning of loading images. For the conversion from RGBA images to grayscale images, within each function, we first index the position of the corresponding pixel by thread id, and then calculate the grayscale value. Every thread within different blocks could compute the grayscale value simultaneously. We apply the same skill to the Sobel operator calculation and conversion from grayscale to RGBA images.

But for the grayscale stretch, in which we did not find an appropriate way to find the maximum value using OpenMP, we do it recursively using CUDA. We first initialize the max value as a copy of the Sobel operator output value and then do a similar reduction operation, with reference to [3]. Intuitively we will find the max value of each block and then find the max value between different blocks, just like different levels. Since within each "reduce" function, we will store the max value at the first position of its local memory, in the end, the very first element of each block will be the max value within this block. Then, for the normalization, we do it also as blocks. Within each block, we use the max value to normalize the value within this block. Hence, we could parallelize the finding max value, normalize the gray value, and further speed up the whole process.

4. Experiment and Result Analysis

4.1 Experiment Design

In this experiment, our input data is an image, so we take the photo of the Sather Gate as an original image with size of 3248 * 2436 pixels. To perform the strong and weak scaling, we also need different sizes of images, so we repeat this photo to build them. In the following section, we will use the Task Name to call these images. Their file names on the GitHub repository and image sizes are listed in Table 1. Besides these photos, we also tested our results on the file called library.png and waikiki.png to make sure we get the edges of input images.

Table 1. Experiment design of task name, input file name and input image size

Task Name	Input file name	Image Size (pixel number)
image	Gate.png	3248 * 2436 = 7,912,128
image 4x	Gate_4.png	6496 * 4872 = 31,648,512
image 9x	Gate_9.png	9744 * 7308 = 71,209,152
image 16x	Gate_16.png	12992 * 9744 = 126,594,048

4.2 Slow Down Comparison Between Serial, OpenMP and CUDA

The time required by the three versions of Sobel algorithm on various problem sizes is shown in Table 2. The problem size is described in section 4.1, with corresponding Task Name.

Table 2. Run times of three implementations of Sobel Edge Detection. (unit: second)

Task Name	Serial	OpenMP(68 threads)	CUDA
image	1.660690	0.079566	0.0011
image 4x	6.626279	0.191321	0.0019
image 9x	14.901165	0.411753	0.0034
image 16x	26.465120	0.724317	0.0055

From Table 2, we can calculate the slowdown of these three implementations. For the serial code, the estimated time complexity is $O(n)$. The slowdown ε calculated through Equation (1) is $-\ 0.002$(compared with $O(n)$ algorithm)

$$\varepsilon(Serial) = \frac{log(26.384604)) - log(1.658905)}{log(16) - log(1)} - 1 = -0.002 \quad (1)$$

For our OpenMP implementation, the estimated time complexity is $O(n^{0.797})$, and slowdown ε calculated through Equation (2) is $-\ 0.203$ (compared with $O(n)$ algorithm) as the problem size increases. For our CUDA implementation, the estimated time complexity is $O(n^{0.580})$, and slowdown ε calculated through Equation (3) is $-\ 0.420$ (compared with $O(n)$ algorithm) as the problem size increases.

$$\varepsilon(OpenMP) = \frac{log(0.724317) - log(0.079566)}{log(16) - log(1)} - 1 = -0.203 \quad (2)$$

$$\varepsilon(CUDA) = \frac{log(0.0055) - log(0.0011)}{log(16) - log(1)} - 1 = -0.420 \quad (3)$$

The change of time across different problem sizes of these two algorithms is illustrated in Figure 2. The serial code was accelerated by OpenMP, and further accelerated by CUDA.

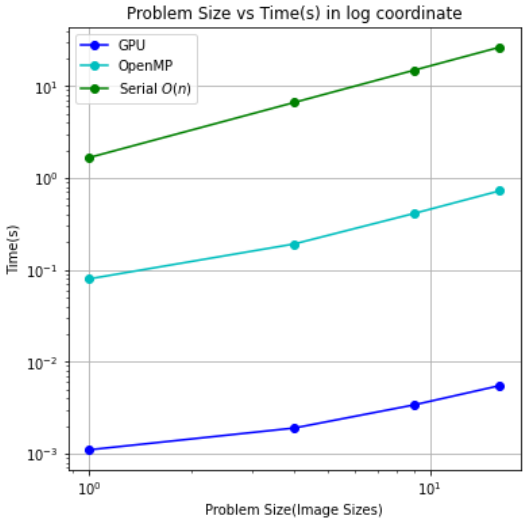


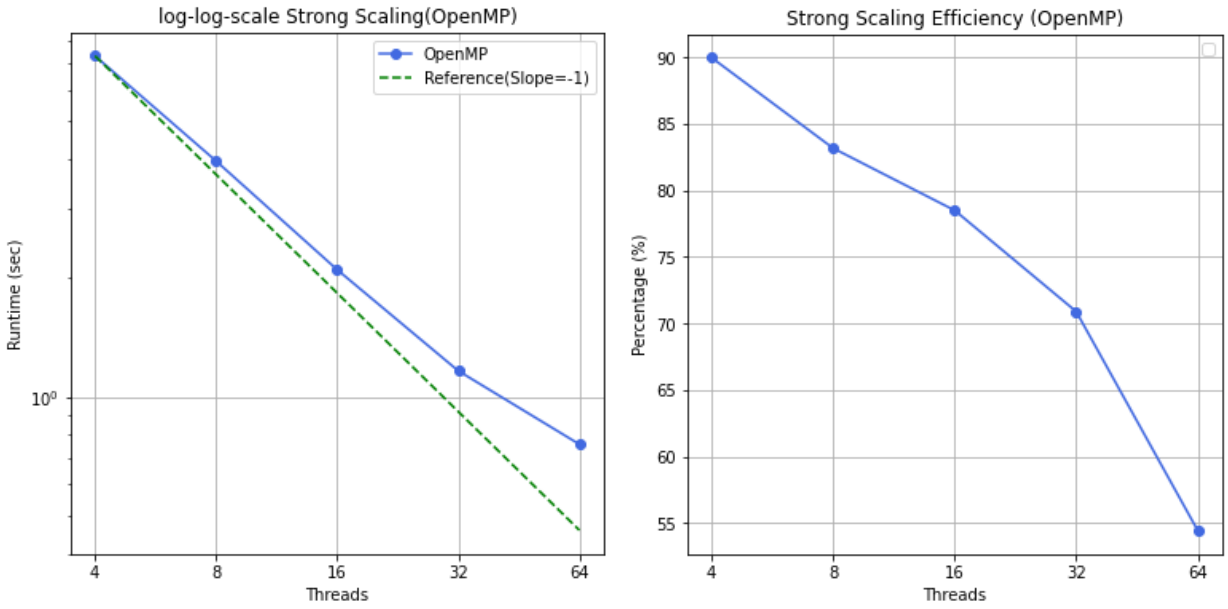
Figure 2. The comparison of time across different problem sizes on serial code and OpenMP and CUDA implementation (log-log)

4.3 Strong and Weak Scaling for OpenMP

4.3.1 Strong Scaling

Table 3. OpenMP Strong Scaling: Time vs. Number of Threads

Task Name	# of Threads	Time(s)	Efficiency
image 16x	4	7.3328	89.95 %
image 16x	8	3.9662	83.15 %
image 16x	16	2.1010	78.48 %
image 16x	32	1.1631	70.88 %
image 16x	64	0.7572	54.44 %



(a) (b)

Figure 3. (a) OpenMP Strong Scaling: Time vs. Number of Threads(log - log). (b) OpenMP Strong Scaling Efficiency: Percentage to ideal vs. Number of Threads(log x axis).

In this part, we want to first analyze strong scaling for the OpenMP version, and since CUDA can not explicitly change the number of threads because this is usually associated with machine settings, we could only report strong and weak scaling for OpenMP, and find the changing trends.

We conducted our experiments using an image whose height is 9744 and width is 12992, for this large image, which approximately reaches the limit for unsigned char type, the serial version of Sobel operator is 26.3846s. We listed all experimental results in Table 3, and to calculate strong scaling, we use the same image and double the number of threads each time. We can see that the strong scaling efficiency for 64 threads is 54.44% as shown in Equation (4), dropping from 70.88% for 32.

$$Efficiency_{strong}(64) = 26.3946 \div (0.7572 \times 64) = 54.44\% \quad (4)$$

Then, we plotted it in Figure 3. For the first plot, we are pretty confident to say our parallelized code has great performance for strong scaling, since it is actually very close to the reference line. The reason why as the number of threads increases, strong scaling is going up is that when we try to find the max value and normalize grayscale value, we could not make it parallel. All other threads need to hold and wait for the main thread to find the max value, and then do the rest of calculations parallel. So with more threads waiting, the strong scaling efficiency will drop just as the second plot suggests. Since other parts of code do not have race conditions, we believe the bottleneck for strong scaling drop significantly in the second part is the max value finding process.

4.3.2 Weak Scaling

Table 4. OpenMP Weak Scaling: Time vs. Image Scale (4 threads per original image size)

Task Name	Threads	Time (second)
image	4	0.458033
image 4x	16	0.534656
image 9x	36	0.602405
image 16x	64	0.750227

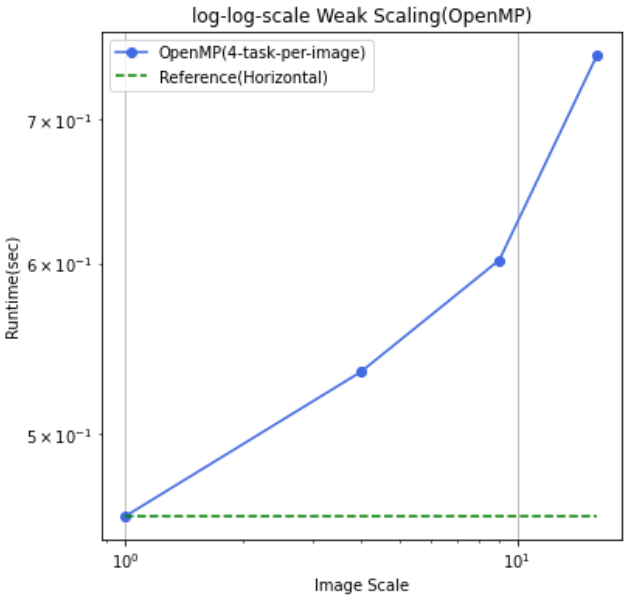


Figure 4. OpenMP Weak Scaling: Time vs. Image Scale relative to original image (log - log)

For weak scaling, we need to change the problem size and number of threads at the same scale each time. In order to do that, we use images whose size is 4, 9, 16 times larger than the original one, and increase the number of threads to the same scale. We collected all the data in Table 4, and plotted it in Figure 4. Ideally, the weak scaling for a perfectly parallelized problem will be a horizontal line, but in practice, since there always remain data dependencies that make some threads need to stop and wait for other threads to finish, lines representing weak scaling will be above the reference line,

meaning that more threads and larger problem set will need to take a longer time. In our case, just as with strong scaling, finding the max pixel value in the whole image holds other threads and we couldn't find a way to parallelize it. So with more threads including, they need to wait for a longer time in summary, leading to our weak scaling going up.

4.4 Runtime Analysis of different parts

4.4.1 How the runtime of different parts changes in different implementations

In this experiment, we timed the cost of each part in Figure 1, and the get global max value part, which was written as a critical section (serial) in OpenMP version and using the max reduction algorithm in CUDA. To keep the timers consistent, we used the omp_get_wtime() function as timer for both serial and OpenMP versions, and used the nvprof command as CUDA timer.

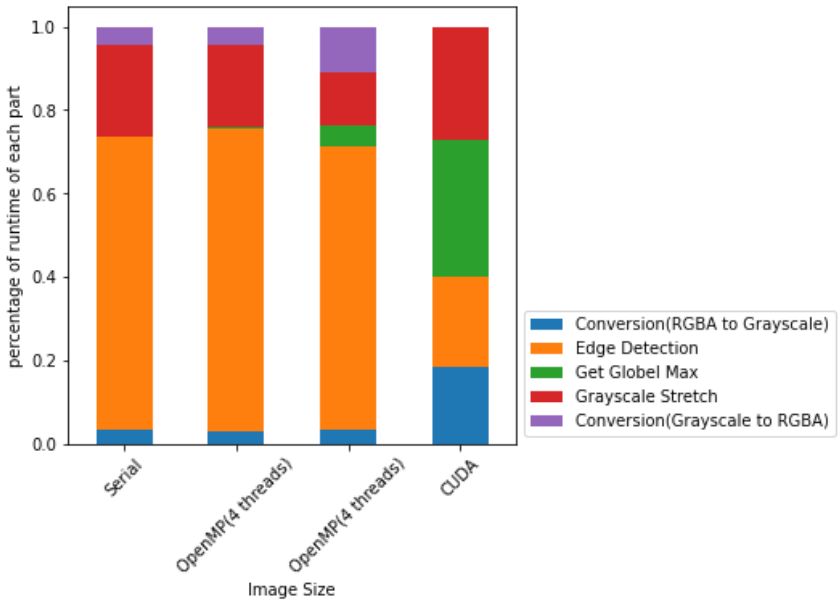


Figure 5. Percentage of each parts runtime comparison

The percentage of time of each part in Serial, 4 threads OpenMP, 64 threads OpenMP, and CUDA is illustrated in Figure 6. Please notice that in CUDA implementation, the conversion of Grayscale to RGBA(color purple) has been a part of Grayscale Stretch(color red) in our implementation, so the red part in the CUDA bar includes these two parts. Based on FIgure 6 we can observe the following findings:

1. The parallelization of Conversion(Grayscale to RGBA) does not perform well in OpenMP with 64 threads. The percentage increases from 4.2% to 10.8% when threads increase from 4 threads to 64 threads. Since this code just contains the scattering of one value to adjacent three memory positions, we think it might be the overhead of OpenMP that leads to this result
2. The serial bottleneck (Get Global Max) in OpenMP is obvious. In the Serial and OpenMP(4 threads), this part occupies 0.13% and 0.49% separately. But in OpenMP(64 threads) it already occupies 5.06%. This value will occupy even more percentage of time when threads number go

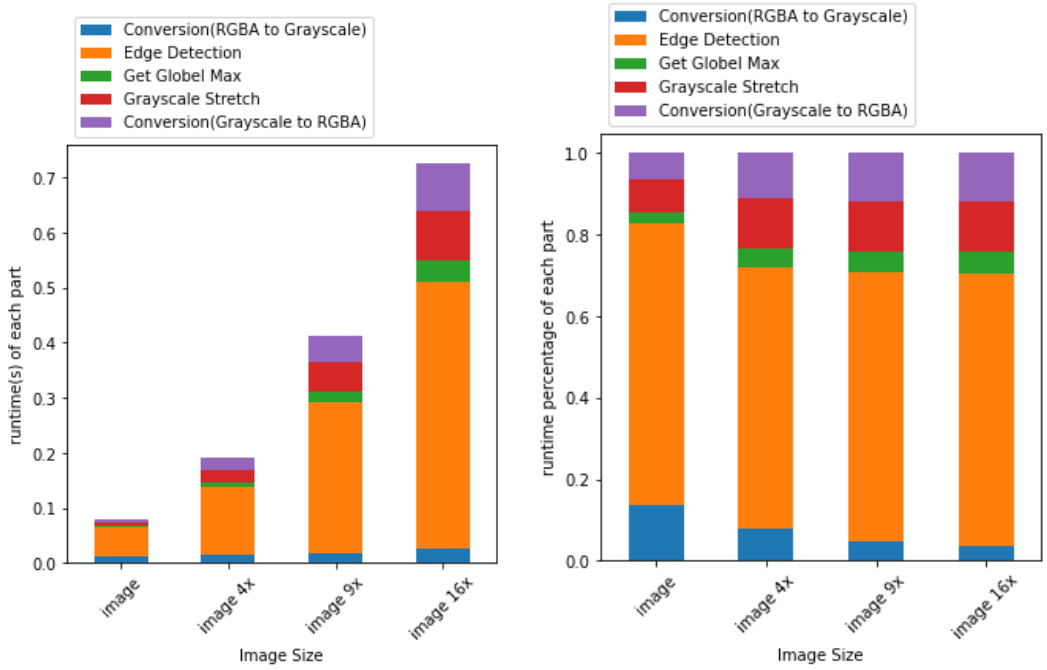
higher. Therefore, in CUDA, we use a max reduction algorithm instead of the naive O(n) solution to get the global max. So even though the NVIDIA V100s GPU has 5120 cores and 80 SM which accelerated the other parts significantly, the Get Global Max part didn't occupy the most percentage of the time.

3. An interesting finding is that, different from Serial and OpenMP, the Edge Detection part didn't occupy the most time in the CUDA version. That may be because the CUDA has special optimization for this kind of convolution operation.

4.4.2 How the runtime of different parts changes for different size of input data for OpenMP

To further figure out how the input data scale (image size, pixel number) influence the runtime(s) and the percentage of runtime in each module, we timed the OpenMP(64 threads) implementation on input images with different sizes. The results are shown in Figure 6. We got the following findings from these figures:

1. The percentage of conversion from Grayscale to RGBA (color purple, which is a scatter) goes higher when the image size becomes larger, while the conversion of RGBA to Grayscale (color blue, which is a gather) goes lower. We think the differences of scattering and gathering may contribute to these differences.
2. When analyzing the serial bottleneck "Get Global Max", we figure out that even the input image expanded seventy-seven percent and the serial part occupation time also expanded 78.9%, the serial part occupation percentage only increased from 5.13% to 5.21%, which is almost the same. We think that may because this proportion is too small, and the fluctuations in other parts lead to insignificant changes in this proportion.



(a) The runtime(s) (b) The percentage of runtime

Figure 6. Change of runtime of different part in OpenMP when change the input data scale

Contribution

For the coding parts, Jingjing and Yaowei took charge of the serial version, Guanyue finished OpenMP implementation, and we worked together on CUDA.

For the report, Jingjing works on Section 1, 2 and 4.2. Guanyue works on Section 3 and 4.3. Yaowei works on Section 4.1, 4.4 and provides figures and experiments results.

Reference

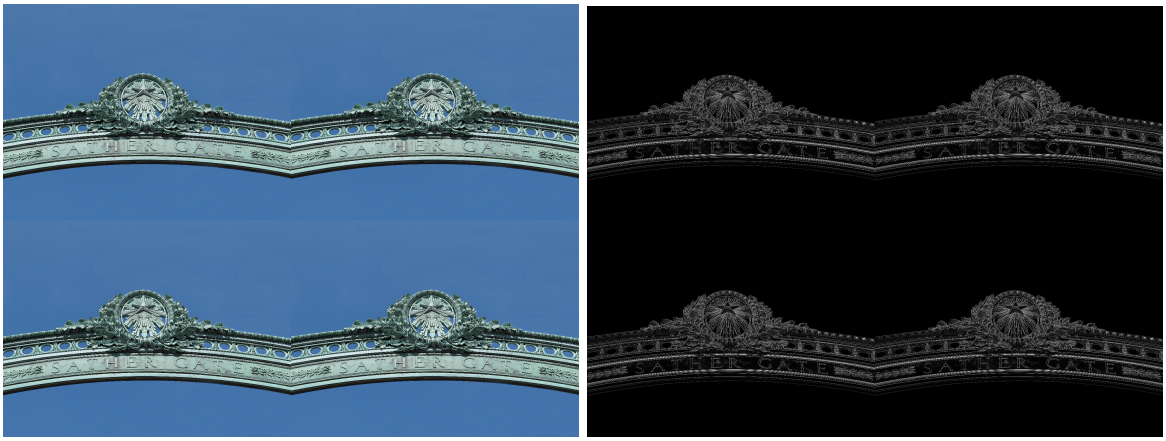
[1] N. Kanopoulos, N. Vasanthavada and R. L. Baker, "Design of an image edge detection filter using the Sobel operator," in IEEE Journal of Solid-State Circuits, vol. 23, no. 2, pp. 358-367, April 1988, doi: 10.1109/4.996.

[2] Lvandeve, “LVANDEVE/Lodepng: PNG encoder and decoder in C and C++.,” *GitHub*. [Online]. Available: https://github.com/lvandeve/lodepng. [Accessed: 10-May-2022].

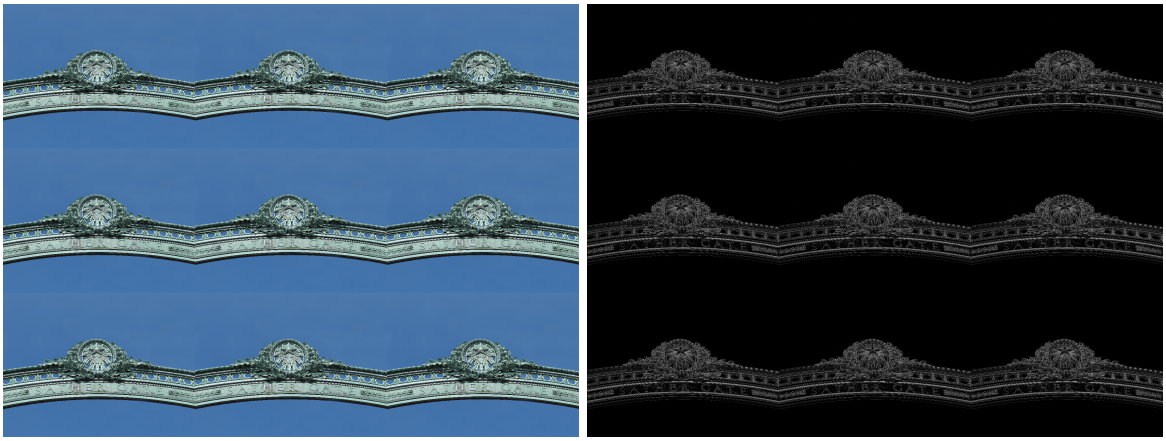
[3] Ricordel, “Ricordel/parallel-sobel: Project for Parallel Computing Course at KTH. in particular, a sobel edge-detector using OpenMP or CUDA.,” *GitHub*. [Online]. Available: https://github.com/Ricordel/parallel-sobel. [Accessed: 10-May-2022].

Appendix

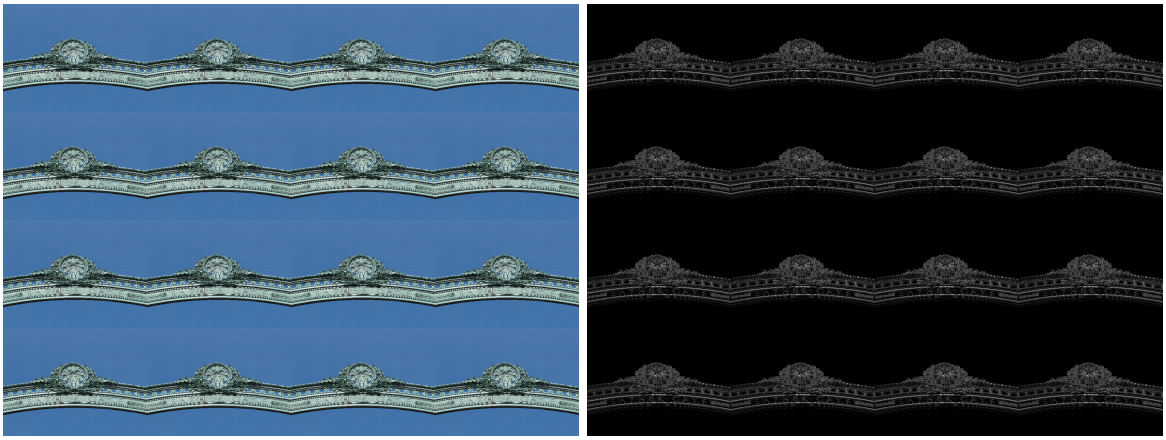
The original RGB input and the edge grayscale output of task: image (Gate.png)



The original RGB input and the edge grayscale output of task: image 4x (Gate_4.png)



The original RGB input and the edge grayscale output of task: image 9x (Gate_9.png)



The original RGB input and the edge grayscale output of task: image 16x (Gate_16.png)