Multicore Computing

Project - Edge Detection Filter

Department of Computer Engineering

Sharif University of Technology

Spring 2022

Lecturer:
Dr. Falahati

Name - Student Number:
Saba Hashemi - 97100581
Amirmahdi Namjoo - 97107212

1 Team Members

- Team member No.1: Saba Hashemi, 97100581
- Team member No.2: Amirmahdi Namjoo, 97107212

2 Introduction

In this project, we implemented an Edge Detection Filter based on Sobel Filter. The project's core is based on SIMD computations using NVIDIA CUDA. The project features both CLI and GUI. The GUI is written in Python, uses the Tkinter library, and communicates with the CUDA core, which is written in C++ using a shell.

3 Implementation

Our project has two main parts. One part is adjusting the contrast and brightness of pictures, and the other part is Sobel Filter.

3.1 Contrast and Brightness

Each pixel's contrast and brightness in an image can be changed using the following formula:

$$\alpha \times \text{pixel} + \beta$$

In which α is contrast and β is brightness. This formula needs to be applied to the whole image to change the contrast and brightness of an image. Due to the parallel nature of the task, it could be easily achieved using CUDA. Our Kernel for this task is as follows:

```
__global__
void adjust(uchar *input,
    uchar *output,
    float alpha,
    float beta,
    size_t size) {
        /* Calculate the index of the pixel that this
        thread is responsible for */
        int idx = blockDim.x * blockIdx.x + threadIdx.x;

        /* Calculate new value of the pixel*/
        uchar pixel = input[idx];
        int value = alpha * pixel + beta;

        /* Clip value to valid range */
        if (value > 255)
            value = 255;
        if (value < 0)
            value = 0;

        /* Store result */
        output[idx] = (uchar) value;
    }
```

3.2 Sobel Filter

There exist multiple filters to implement edge detection. For this project, we used Sobel Filter. To use Sobel Filter, the whole image A is convolved with two matrices G_x and G_y, which are shown below:

+1	0	-1
+2	0	-2
+1	0	-1

Table 1: G_x

+1	+2	+1
0	0	0
-1	-2	-1

Table 2: G_y

Then, the gradient magnitude matrix is constructed by calculating the $\sqrt{g_x^2 + g_y^2}$ for all elements of G_x and G_y. The resulting matrix $G = \sqrt{G_x^2 + G_y^2}$ is used to show edges of the picture.

The kernel code for this part is as follows:

```
__global__
void conv(uchar *input,
    uchar *output,
    float *kernel_v,
    float *kernel_h,
    size_t rows,
    size_t cols,
    size_t kernel_size) {
    /* Calculate the index of the pixel that this
    thread is responsible for */
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int i = idx / cols;
    int j = idx % cols;

    float sum_v = 0, sum_h = 0;
    /* Range over kernel entries */
    for (int kidx = 0; kidx < kernel_size * kernel_size; kidx++) {
        int ki = kidx / kernel_size;
        int kj = kidx % kernel_size;

        int new_i = i + ki - kernel_size / 2;
        int new_j = j + kj - kernel_size / 2;

        /* Pass pixel if we are out of boundry */
        if (new_i < 0 || new_i >= rows || new_j < 0 || new_j >= cols)
```

```
            continue;

        int new_idx = new_i * cols + new_j;

        /* Convolve each of horizontal and vertical filters
        with image */
        sum_v += kernel_v[kidx] * (float) input[new_idx];
        sum_h += kernel_h[kidx] * (float) input[new_idx];
    }

    /* Calculate new value of the pixel */
    int value = (int) sqrt(sum_v * sum_v + sum_h * sum_h);

    /* Clip value to valid range */
    if (value > 255)
    value = 255;
    if (value < 0)
    value = 0;

    /* Store result */
    output[idx] = (uchar) value;
}
```

3.3 Memory Management

When calling CUDA functions, we need to ensure all arguments are stored in CUDA-accessible memory. For this reason, we used `cudaMalloc` function to allocate GPU memory and `cudaMemcpy` to copy data from host memory to device and visa-versa. Also, `cudaFree` is used to free GPU memory.

An example from the code is shown below:

```
/* Allocate GPU memory for input and ouput image */
cudaMalloc(&input_image_device, total_size_bytes);
cudaMalloc(&output_image_device, total_size_bytes);

/* Copy input image to GPU memory */
cudaMemcpy(input_image_device, input_image_host, total_size_bytes,
    cudaMemcpyHostToDevice);

/* Do some work ... */

/* Copy output image to host memory */
cudaMemcpy(output_image_host, output_image_device, total_size_bytes,
    cudaMemcpyDeviceToHost);

/* Free allocated memory */
cudaFree(output_image_device);
cudaFree(input_image_device);
```

4 Results

To compare the speedup from switching to GPU, we tested five different images (included in `benchmark` folder in `core`). The dimensions of these five images are `1920x1080` (Full HD), `1920x1200`, `1920x1440`, `3840x2160` (4K), `7680x4320` (8K). As our code includes two parts, namely "Brightness and Contrast Adjustment" and "Sobel Edge Filter," we recorded the timing details of each of these parts separately. We plotted different graphs for each part and the total time of running these parts. Note that the time needed to load the picture using OpenCV and write it to the file is not included in the data.

For the GPU implementation part, we measured time for three different block sizes, "256", "64", and "16".

It is noteworthy that because the runtime of CPU implementation is orders of magnitude higher than GPU code, we plotted two different plots for each part; one includes CPU timings, and the other just includes GPU timing have better resolution.

Also, the values for Brightness and Contrast adjustment in all runs are $\alpha = 0.8, \beta = 50$.

You can view the results in the charts below:

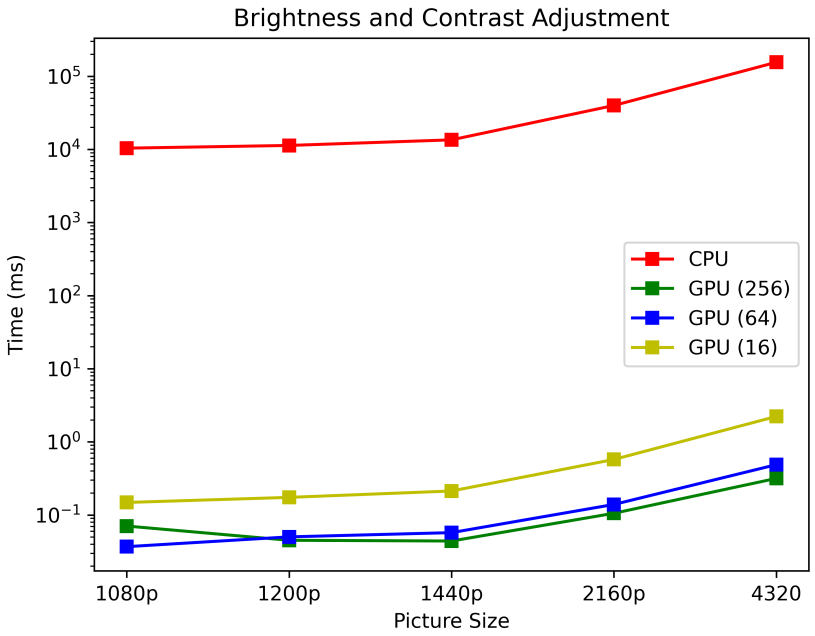

Figure 1: Brightness and Contrast Adjustment Benchmark (Including CPU)

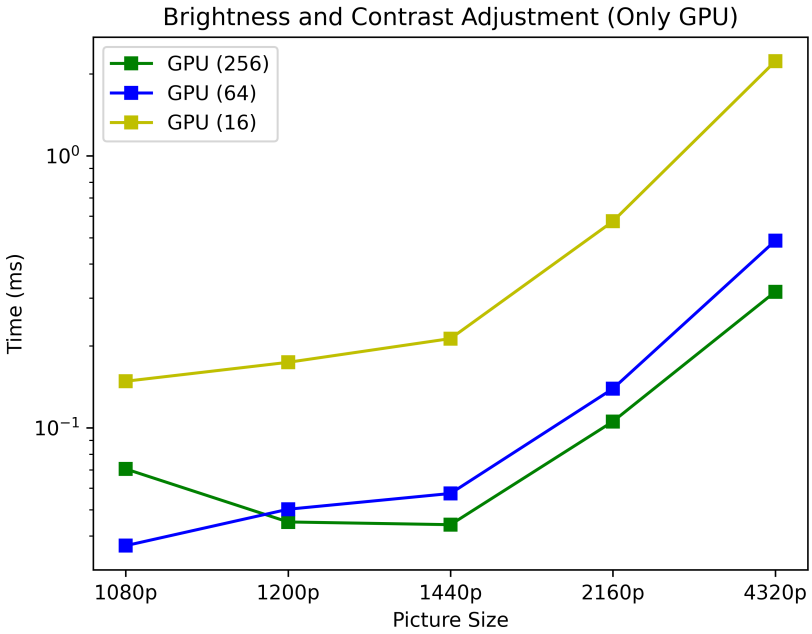

Figure 2: Brightness and Contrast Adjustment Benchmark (Without CPU)

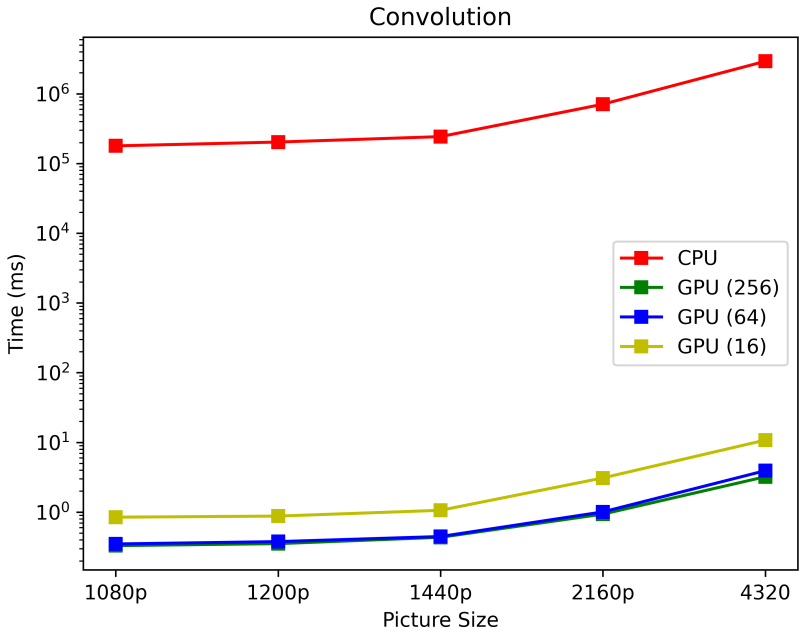

Figure 3: Sobel Edge Detection Filter Convolution Benchmark (Including CPU)

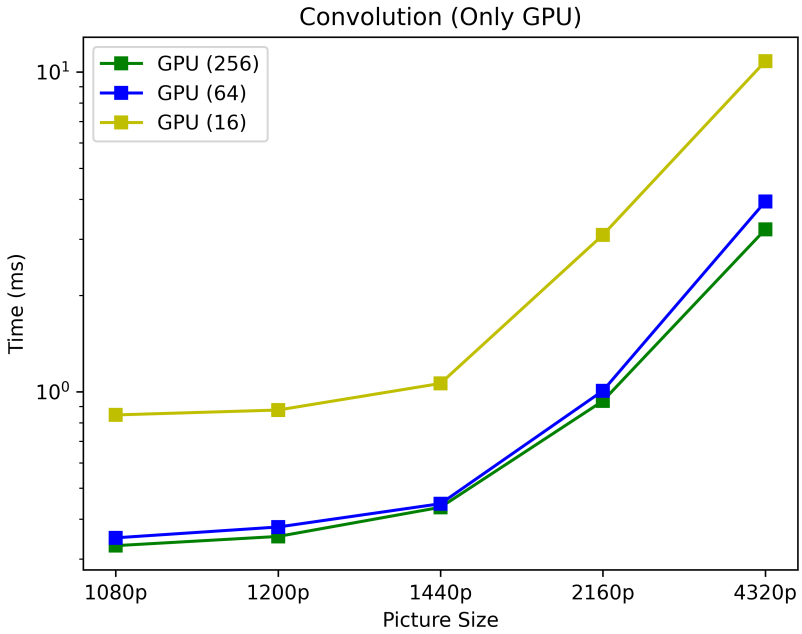

Figure 4: Sobel Edge Detection Filter Convolution Benchmark (Without CPU)

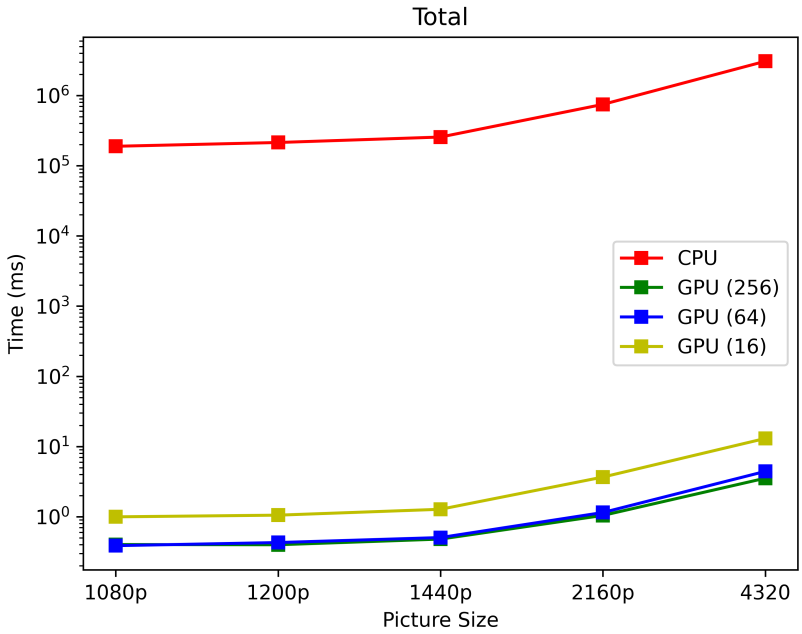

Figure 5: Total Runtime Benchmark (Including CPU)

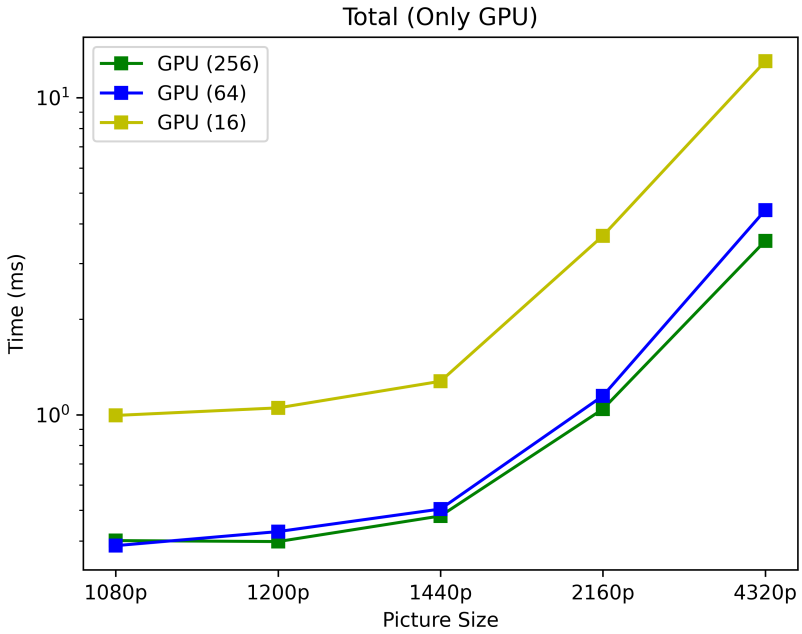

Figure 6: Total Runtime Benchmark (Without CPU)

As evident from the plots, we gain a huge exponential speedup by using GPU and CUDA. Also, in most cases, ”256 threads per block” was the best option. While, in most cases, the difference between 256 and 64 is minor, the difference between 16 and 256 is substantial.