

# **CS406 Project – Final Report**

## **Parallelized Digital Image Processing**

**Ege Şenses 23570, Berkay Demireller 23810, Selen Özcan 23914**

GitHub link: <https://github.com/brkd/ImageProcessing406>

### **1. Problem Definition**

Digital Image Processing refers to the set of algorithms written for making various operations on images, by using the digital representations of said images. Images are kept in a matrix format where each pixel is located in one cell, and carries an integer that decides its intensity. The pixels can be encoded for different color models like grayscale, RGB or CMYK. Every task done for image processing is fundamentally just some calculations, ranging from simple to very complex, on these matrices and pixel values.

Digital Image Processing has been developing since the 1960s and has become the norm for applying virtually any change or operation onto an image, be it color processing, image recognition or image compression. Its rise and progress have been mostly defined by the developments in Computer Science and Discrete Mathematics, as well as the increase in demands in sectors like environmental science, military, medical science and even agriculture.

The domain of applications for efficient Digital Image Processing methods have grown substantially in recent history, thus we believed that working on the parallelization of said methods constituted a goal both significant and large in scope.

### **2. Need for Parallelization**

Image Processing is done with excessive loops that iterate over every single pixel, or in some cases, small batches of pixels. This means that any algorithm used for manipulating images is quite suitable for parallelization. As the resolution of an image increases, so does its size; which means that the need for high performance computing also increases due to how we'll have more iterations and actions per loop.

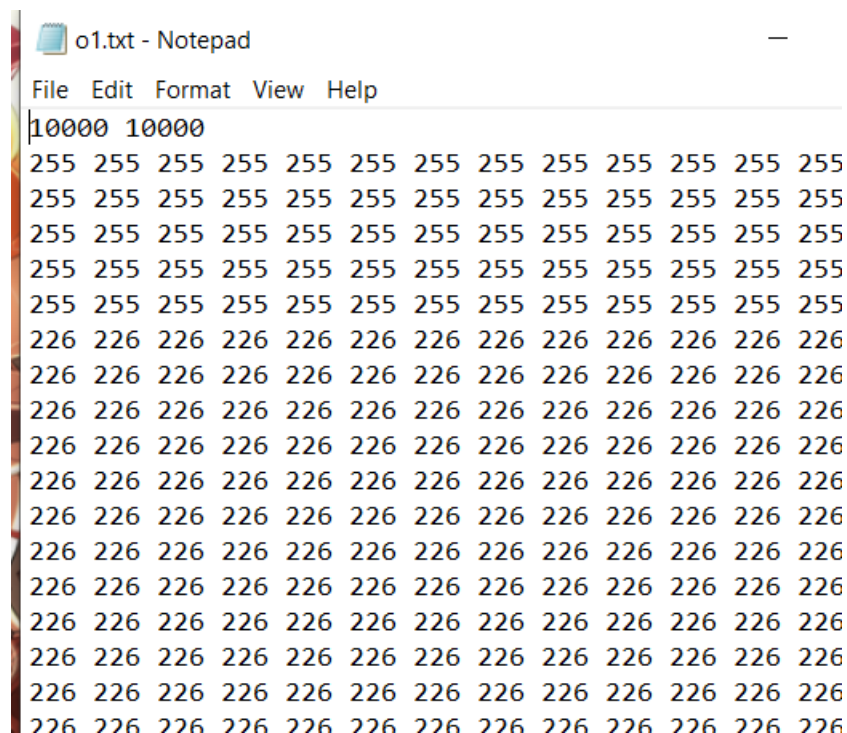
An example of an image processing algorithm that is amply eligible for parallelization is image warping. Image warping refers to significantly distorting the shapes present in an image or video; which is usually done for correcting image distortion or morphing. It is done by applying a transformation on every pixel and replacing the new pixel values with the result of the transformation, which is actually a technique seen regularly in Digital Image Processing.

We have parallelized multiple Digital Image Processing algorithms including Linear Scaling, Median Filter, Sobel Filters, Sobel Edge Detection, Grayworld, Ordered Dithering, Vertical Mid-axis Reflection and Rotations of varying degrees. And we believe we have been able to show that the run-time taken by these processing algorithms have been significantly reduced by employing parallel programming.

### 3. Data Summary

The data that we've worked on, in short, is composed of matrices with 8-bit values that represent the brightness of a pixel in grayscale. The values a pixel can take range from 0 for pitch black to 255 for completely white.

All of our algorithms work on .txt files, so we transpose .jpg files into .txt before any processing is done. The outputs are also given in .txt file, which we convert back into .jpg. The .txt file starts with the dimensions of the image and then lists the values of all of the pixels in 8-bit grayscale:



The screenshot shows a Notepad window titled 'o1.txt - Notepad'. The menu bar includes 'File', 'Edit', 'Format', 'View', and 'Help'. The text content of the file is as follows:

```
10000 10000
255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
226 226 226 226 226 226 226 226 226 226 226 226 226
```

*Fig. Image converted to .txt file example*

The algorithms are then free to traverse these files however they need to in order to create their results; some read pixel by pixel whereas others read small KxK blocks at once.

Luckily, there is an abundance of images available on the net for the purposes of testing, and thus acquiring the test data was not a challenge in any way. In fact, if there was a need for having an image with very specific dimensions, one could simply create said file on their own

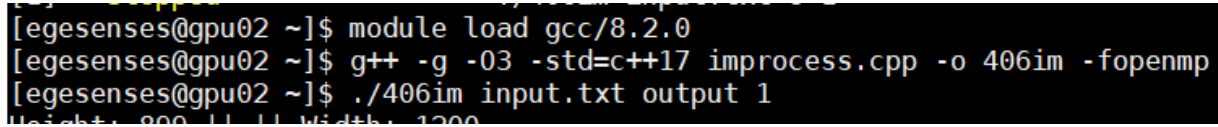
computer with the right amount of pixels and work on that. We've worked with images with sizes ranging from 900x1200 to 10000x10000 and managed to get correct results on all.

#### 4. The Application

The program takes an image as its input in .txt format, applies multiple image processing algorithms onto it and prints the runtime and the result images. It does so for the sequential, CPU-parallel and GPU implementations of the algorithms so that one can compare the results. Python scripts are used for converting .txt and .jpg files into one another.

The main program reads the grayscale data from the given.txt file and creates a 1-dimensional, “flattened” data structure for keeping the image's information. After that, this array is fed into multiple algorithms and displays the run-times. If the user demands it, it prints out their outputs as well.

An example of how we run the CPU code is given below:



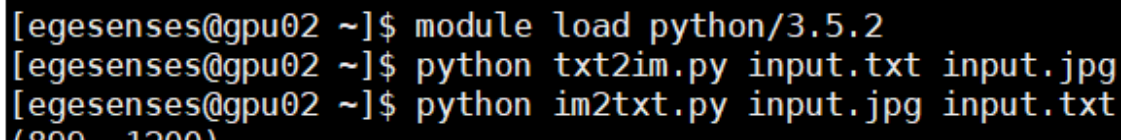
```
[egesenses@gpu02 ~]$ module load gcc/8.2.0
[egesenses@gpu02 ~]$ g++ -g -O3 -std=c++17 improcess.cpp -o 406im -fopenmp
[egesenses@gpu02 ~]$ ./406im input.txt output 1
Height: 800 Width: 1200
```

*Fig. Running the CPU code*

Notice that when running the executable, 3 things are given as input:

- An input .txt file that was converted from a .jpg file
- The name for the output files
- 1 for printing output, 0 for skipping it

Also, the Python scripts used for converting files are called “txt2im.py” and “im2txt.py”.



```
[egesenses@gpu02 ~]$ module load python/3.5.2
[egesenses@gpu02 ~]$ python txt2im.py input.txt input.jpg
[egesenses@gpu02 ~]$ python im2txt.py input.jpg input.txt
(800, 1200)
```

*Fig. Converting .txt and .jpg files with Python scripts*

We explain how exactly to run the code at the end of the report.

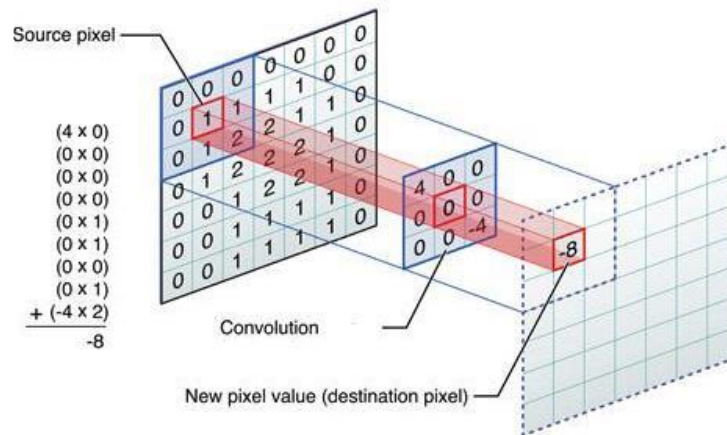
Loop parallelism is employed over the “for loops” in the algorithms we've written. SIMD versions have been created for most of these algorithms for a vectoral approach as well. Additionally, all of these algorithms except Sobel Edge Detection have also been implemented with CUDA for working with GPUs. As one can see from the test results given further down, although CPU parallelism does provide speed-ups, the GPU variants are by far the fastest versions.

Before moving onto the algorithms, one should make note of what is possibly the most important technique in all of Digital Image Processing, which is Convolution. It operates over

almost all kinds of digital signals; speech, image or video. It is also employed for creating a relation between the input and the output of linear systems.

Convolution creates a new set of pixels from the previously given one by applying linear operations. These linear operations are enforced with a window, also called a Kernel, whose values and size are determined beforehand. A visual way of describing Convolution is imagining putting the Kernel onto the input matrix and multiplying the numbers that overlap before adding them all together. The center value of the Kernel coincides with the source pixel for the calculation. This pixel is then replaced with a weighted sum of itself and neighbors.

Different Kernels can be used for achieving different results. For instance, an Identity Kernel that consists of all 0s and only a single 1 in the center does not alter the image in any way. A Kernel where the center value is too high and the neighbors are close to 0 would apply a sharpening filter. A Gaussian Blur Kernel, where the center receives the biggest value and the surrounding values diminish the more they're further from the center, would cause a blurring effect (Ahn, 2018). The application of the Kernel can be given as follows:



*Fig. Convolution*

#### 4.0. Parallelization and GPGPU

As it was discussed before, image processing algorithms are matrix operations in nature. For loops are used in high numbers for almost all image processing algorithms and this makes them an easy candidate for parallelization and GPGPU. Since these are simple algorithms, our OpenMP implementations consist of loop level parallelizations and certain private parameters. CUDA implementations make use of both threads and blocks, and it also uses shared memory for median filtering since window usage is very suitable for shared memory. The current CUDA implementation is suitable for larger images because it uses low amount of blocks and high amount of threads. To get better output images with smaller images (especially for median filtering) the block number could be decreased. Thread numbers should be adjusted accordingly if block numbers are decreased, since each pixel index should be accessible using a combination of block and thread IDs. The specifications are left as comments in the source code.

## 4.1. Linear Scaling

This algorithm receives an image in grayscale and finds the minimum and maximum values inside it (in our case, we do so as a part of preprocessing and feed the minimum and maximum values as input), call them  $x_{\min}$  and  $x_{\max}$ . Afterwards, it scales  $x_{\min}$  to 0 and  $x_{\max}$  to 255, causing every pixel to undergo a similar change. We formulate this as follows:

$$g(x) = \frac{255}{x_{\max} - x_{\min}} (x - x_{\min})$$

Linear Scaling helps find a balance in the brightness/contrast of an image. A mostly dark image would have a tiny difference between  $x_{\min}$  and  $x_{\max}$ , which means inflating the difference to the maximum possible value of 255 helps brighten the image:



*Fig. Linear Scaling example*

## 4.2. Median Filtering

This algorithm is regularly used for noise reduction (Huang et al., 1979). Given a window size  $K$ , it looks up the  $K \times K$  matrix around a pixel, finds the median value and replaces the pixel in the center with this median value. Notice that, much like how the Kernel operated in Convolution, the source pixel is fixated in the center of the  $K \times K$  window. The window size we used is 3.

Median Filtering usually acts as a pre-processing method before further applications that are very susceptible to noise, like edge detection. Such algorithms amplify the noise greatly, which is why removing said noise beforehand is mandatory.

One should note that although the edge preservation property of Median Filtering is quite preferable in this regard, it runs into a lot of issues on the boundaries of the image. This is because with pixels on the boundary, there are fewer neighbors to process. Especially in corners, our  $3 \times 3$  window would have only 3 neighbors to look at instead of the regular 8. Hence noise removal from boundaries is not always possible. Repeating the source pixel's value in the missing values' stead

is one possible workaround. Changing the shape of the window slightly for the boundaries is also a possibility.

Here's how Median Filtering removes salt-and-pepper noise:



*Fig. Median Filtering example*

One can observe that the noise is greatly reduced, however a considerable amount of noise still remains on the boundaries.

### **4.3. Sobel Filtering**

Named after Irwin Sobel, this method consists of two convolutions, one in the horizontal axis and the other on the vertical. It produces a crude gradient approximation that emphasizes the edges so that one can later feed its input into an edge detection algorithm.

Visually, the edges will appear much brighter whereas the rest will appear very dark. The computations of this algorithm is relatively inexpensive since it is practically two separate convolutions.

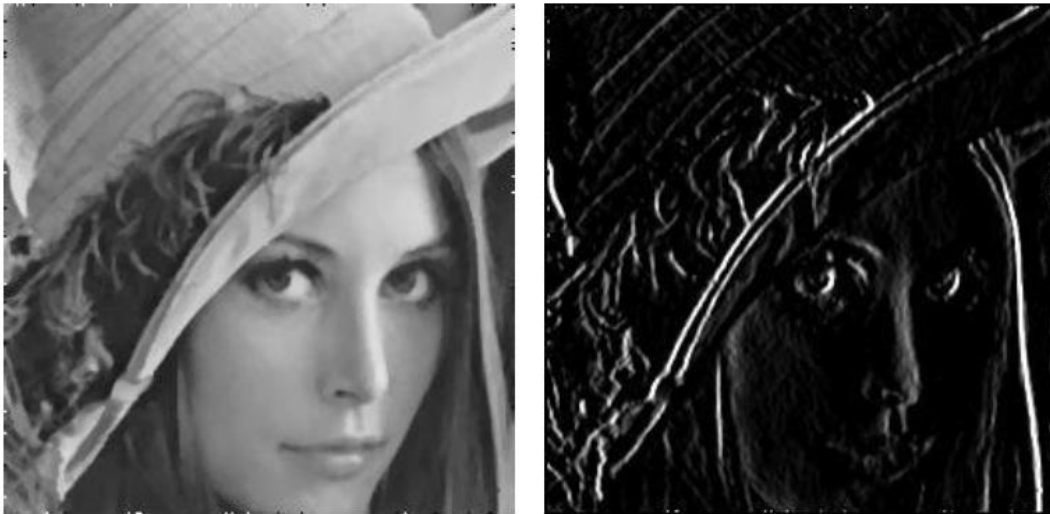
The edge size for the Kernel is 3x3. The Kernels for Sobel Filtering are regularly in the following form:



$$\begin{bmatrix} a & 0 & -a \\ b & 0 & -b \\ a & 0 & -a \end{bmatrix} \quad \begin{bmatrix} a & b & a \\ 0 & 0 & 0 \\ -a & -b & -a \end{bmatrix}$$

Notice that they're practically the same Kernel, given in both vertical and horizontal configurations. The most commonly used Kernel has  $a = 3$  and  $b = 10$ . The convolution operation is done entirely in the same way as what has been described before.

An example of Sobel Filtering is given below:



*Fig. Sobel Filtering example*

#### **4.4. Sobel Edge Detection**

Following Sobel Filtering, one can take these filtered images and feed them into Sobel Edge Detection. The output will have each edge outlined with bright pixels whereas the rest is pitch black, any other detail is basically lost. In applications regarding robotics and computer vision, edge detection algorithms enable machines to “understand” and interpret what they see.



*Fig. Sobel Edge Detection example*

#### 4.5.GrayWorld

This algorithm is commonly used for illuminant estimation. The premise is that, if a photo is color-balanced, the average of all the colors is neutral gray. Hence, one can calculate the average pixel value of an image and then calculate a scaling value by calculating the ratio between 127.5 (which is the expected average in a picture that has every shade equally) and the actual average of the image. This scaling value is then used to alter every single pixel. Do note that, since the algorithms we're using deal with 8-bit grayscale values, the pictures are always in black and white. This does not stop us from implementing a Gray World function and testing it, however the changes it does is not apparent in the final visual result produced since it is in black and white anyway.

In the example given below in color, one can see that Gray World estimates the actual illuminant of the environment and changes the image to match that:



*Fig. GrayWorld example*



## 4.6. Reflection

We've scrapped other variants of reflecting algorithms since they fundamentally do the same work. This method places a vertical axis right in the middle of the image and computes the reflection based on that. It is also quite easy to alter this code to take a complete reflection rather than just half, as well as creating a horizontal reflection rather than a vertical one.



*Fig. Reflection example*

## 4.7. Ordered Dithering

Image Dithering is intentionally applying noise unto an image in the hopes of randomizing quantization error so that color banding or other large patterns can be avoided. Ordered Dithering is the variant that follows a pre-determined path. For each pixel, Ordered Dithering offsets its color by using a “threshold map”:

$$\begin{bmatrix} 128 & 0 \\ 192 & 64 \end{bmatrix}$$

This matrix is placed over 2x2 areas of pixels, and the pixels that have larger values than the thresholds are maximized to 256 whereas the others are minimized to 0. Do note that for very large images, the threshold map's size might need to be increased for better observing the results visually.

Ordered Dithering is usually employed for reducing the color depth of images. Its computations result in a so-called “crosshatch” pattern.

An example is given below:



*Fig. Ordered Dithering example*

#### 4.8. Rotations

Lastly, we've implemented rotations of degrees  $90^\circ$ ,  $180^\circ$  and  $270^\circ$ . Do notice that rotations of degrees  $90^\circ$  and  $270^\circ$  switch the dimensions of the image.

One can apply the following changes for rotating an image:

- For rotating  $90^\circ$  degrees,  $\text{image}[i][j] \rightarrow \text{image}[\text{width}-j-1][i]$ .
- For rotating  $180^\circ$  degrees,  $\text{image}[i][j] \rightarrow \text{image}[\text{height}-i-1][\text{width}-j-1]$ .
- For rotating  $270^\circ$  degrees,  $\text{image}[i][j] \rightarrow \text{image}[j][\text{height}-i-1]$ .



*Fig. Rotation  $180^\circ$  example*

All of these algorithms also have parallel versions and most have additional SIMD versions implemented. The GPU versions of these algorithms have also been implemented and tested, and yield the best runtimes.

## 5.Results

CPU results:

A	B	C	D	E	F
3456x5184	2	4	8	Sequential	SIMD
Linear Scaling	0.0146162	0.0131276	0.0125457	0.0416458	0.0132143
GrayWorld	0.0205796	0.0193217	0.021817	0.0448715	0.0451328
Reflection	0.0139852	0.0133422	0.0126407	0.0421824	0.0422297
Ordered Dithering	0.0145684	0.0128582	0.0124659	0.0480085	0.0478448
Rotation 90	0.0679653	0.0625284	0.0566692	0.284713	0.061816
Rotation 180	0.127155	0.109717	0.100601	0.556695	0.108193
Rotation 270	0.0685107	0.0647423	0.0594437	0.273651	0.0616885
Median Filter	0.0361159	0.0348756	0.0356335	1.77474	0.0213309
Sobel X	0.0178218	0.018529	0.0181815	0.0669244	0.066108
Sobel Y	0.0165257	0.0156092	0.0163777	0.066108	
Sobel Edge	0.0169514	0.0159364	0.0153116	0.0910353	
10000x10000	2	4	8	Sequential	SIMD
Linear Scaling	0.0738723	0.0586229	0.0645616	0.22816	0.0635805
GrayWorld	0.08279	0.0650957	0.0840117	0.248804	0.252904
Reflection	0.074013	0.0583507	0.065051	0.232509	0.232011
Ordered Dithering	0.0727366	0.0578218	0.0636794	0.254467	0.259913
Rotation 90	0.166381	0.148096	0.178345	1.83597	0.156247
Rotation 180	0.310238	0.282964	0.364592	3.67159	0.309012
Rotation 270	0.156142	0.142561	0.183272	1.83645	0.157525
Median Filter	0.0628404	0.0583405	0.0918858	3.09128	0.0658029
Sobel X	0.0770428	0.0592741	0.0711626	0.366354	
Sobel Y	0.0751184	0.0592866	0.0709141	0.479563	
Sobel Edge	0.106566	0.0694916	0.0655003	0.49594	

The results for the GPU variants are given below:

Algorithms	1440x2560	3000x4000	3456x5184	6675x10667	10000x10000
Linear Scaling	0.000447539	0.001066183	0.001432934	0.005104413	0.006807726
GrayWorld	0.000270035	0.000818861	0.001220353	0.004966005	0.00746464
Reflection	0.00028023	0.000879638	0.00137225	0.005300665	0.00741665
Ordered Dithering	0.000294576	0.000881306	0.001293999	0.005410259	0.00744828
Rotation 90	0.000576864	0.001792217	0.00324351	0.01117047	0.01507945
Rotation 180	0.000277434	0.000815789	0.001218074	0.004950199	0.006838198
Median Filter	0.001146242	0.00366759	0.005740345	0.02283644	0.02622359

*Fig. GPU Results Average*

These are the average of 10 tests, and are measured in seconds.  
Enrollment in local colleges, 2005

*Fig. CPU Sequential Results*

## 6. How to Run the Code

GitHub link: <https://github.com/brkd/ImageProcessing406>.

One can find all of the mentioned algorithms and scripts on the GitHub page, however we'll also list the links for each important page for convenience:

CPU Code: <https://github.com/brkd/ImageProcessing406/blob/master/improcess.cpp>

GPU Code: <https://github.com/brkd/ImageProcessing406/blob/master/GPU/gpimprocess.cu>

txt2im.py script: <https://github.com/brkd/ImageProcessing406/blob/master/txt2im.py>

im2txt.py script: <https://github.com/brkd/ImageProcessing406/blob/master/im2txt.py>

Some test images: <https://github.com/brkd/ImageProcessing406/tree/master/img>

Like described above, we compile the code with the following command:

```
g++ -g -O3 -std=c++17 improcess.cpp -o 406im -fopenmp
```

Once a .jpg file is chosen for input, one can turn this file into a .txt file with the “im2txt.py” script with the following command:

```
python im2txt.py input.jpg input.txt
```

Note that the names do not have to be “input.jpg” or “input.txt”. Then, one can run the executable with the following command:

```
./406im input.txt output 1
```

The input.txt is the input file, “output” is the name to be given to the output files and “1” means that we want the output printed out.

If 1 was selected instead of 0, we can again use a python script, this time “txt2im.py”, for converting this .txt file into a .jpg file with the following command:

```
python txt2im.py input.txt input.jpg
```

```
[egesenses@gpu02 ~]$ module load gcc/8.2.0
[egesenses@gpu02 ~]$ g++ -g -O3 -std=c++17 improcess.cpp -o 406im -fopenmp
[egesenses@gpu02 ~]$ ./406im input.txt output 1
Height: 800 Width: 1200
```

```
[egesenses@gpu02 ~]$ module load python/3.5.2
[egesenses@gpu02 ~]$ python txt2im.py input.txt input.jpg
[egesenses@gpu02 ~]$ python im2txt.py input.jpg input.txt
(800, 1200)
```

*Fig. How to run the code*

For the GPU code, we run the following command:

```
nvcc -o gpu gpimprocess.cu
```

Note that this can yield a “string to char” warning, but you can safely ignore it.

```
[egesenses@gpu02 ~]$ module load gcc/5.3.0
[egesenses@gpu02 ~]$ module load cuda/10.0
[egesenses@gpu02 ~]$ nvcc -o gpu gpimprocess.cu
```

*Fig. How to compile the GPU code*

Then, assuming the input file has been converted to a .jpg file, one can run the code with the following command:

```
./gpu input.txt 1
```

“input.txt” is the input file and 1 means “print the output” whereas 0 would mean the opposite.

```
[egesenses@gpu02 ~]$ ./gpu input.txt 1
```

*Fig. How to run the GPU code*



## 7. Conclusion and Future Work

Due to the nature of 2D images and their representation, image processing algorithms are very suitable to parallelization and GPGPU programming. Our project intended to show that using parallel algorithms most image processing algorithms can achieve considerable speed-ups and to a certain extent succeeded to do so. Some of the algorithms described above behaved contrary to our expectations and resulted in low speed-ups or even in longer execution times. Certain ideas were discussed among team members to improve the implementations so that these algorithms would also have lower execution times but we couldn't complete the discussed implementations and thus couldn't achieve better execution times. A future iteration of the project could include those implementations as well. CUDA implementations surpassed all of the parallel implementations as expected but could be further improved upon with more proper usage of shared memory. Some algorithms were not included in CUDA implementations but they can be easily implemented as well, which can also be included in future iterations. CUDA supports video operations as well and video processing is widely used as it's one of the fundamental parts of robotics. Video processing algorithms such as velocity calculation of moving objects could be implemented with CUDA as well in the future iterations, if possible. Our implementations did not include any global operations on images, if possible to be used with CUDA the set of global operations can result in significant speed-ups as there are a lot of expensive global image processing operations.

## 8. References

- Ahn, S.H. (2018). "*Convolution*". Retrieved from: [http://www.songho.ca/dsp/convolution/convolution.html#convolution\\_2d](http://www.songho.ca/dsp/convolution/convolution.html#convolution_2d)
- Fig retrieved from <https://www.mathworks.com/help/images/ref/illumgray.html>
- Fig retrieved from <https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411>