# General-purpose Computing on Graphics Processing Units for Real-time Analysis of Scanning Electron Microscope Images

Liuchuyao Xu

May 23, 2020

## Abstract

# Chapter 1

# Introduction

The scanning electron microscope (SEM) is a type of microscope that produces images using signals generated from the interaction between electrons and the surface under observation. It has higher resolutions than traditional optical microscopes—an SEM can have a resolution lower than one nanometre, whereas that of an optical microscope is limited to a few hundred nanometres. This has benefited a variety of fields by allowing scientists to see micro-details of objects that were previously impossible to observe. For example, the SEM can be used to study structures of semiconductor devices [1] and to view changes in bacterial cells [2].

Fig. 1.1 illustrates how an SEM works. The electron gun generates an electron beam, which is transformed into an electron probe after passing through the condenser lens and objective lens. It is then scanned across the specimen under the effect of the scanning coil. As a result of the interaction between the incident electrons and the specimen, some electrons (which are called secondary electrons) are emitted from the specimen. The detector collects the secondary electrons and generates signals based on their energy levels. The display unit uses the signals to produce one image after each complete scan of the specimen.

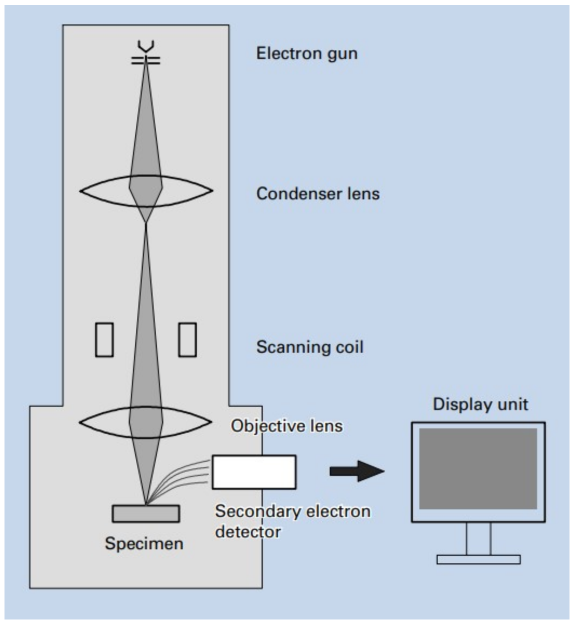


Figure 1.1: Basic construction of an SEM [3].

Many image analysis methods have applications in the field of SEM, and the fast Fourier transform is an especially popular one since it can be used to evaluate the focusing and astigmatism of an SEM [see section **??**]; however, due to the complexity of the algorithm and a lack of fast hardware, real-time analysis had been either impossible or impractical in the past. In 1997, with an advanced central processing unit (CPU)—the Pentium Pro, it was only possible to achieve a refresh rate of 0.6 frames per second for 8-bit 1024 × 1024 input images [4]. Although the enhancement of CPUs has enabled faster computations throughout the years, what really brings the speed to a different level is the development of graphics processing units (GPUs).

The GPU used to be a highly specialised hardware that was designed to excel in rendering complex, high-resolution, real-time 3D scenes for games; however, its special architecture has made it outperform CPUs in many other areas and resulted in the birth of the idea of general-purpose computing on graphics processing units (GPGPU) [5], where GPUs are used to perform computations that are traditionally handled by CPUs.

A key characteristic of GPUs is massive parallelism. Depending on their positions, pixels in a 3D scene often require different processing to achieve effects such as lighting, blurring, and fogging. GPUs do this by breaking down the scene into fragments and manipulating each fragment individually. Modern GPUs have thousands of parallel processor cores each running tens of parallel threads to meet the the high requirement for parallelism. For example, the NVIDIA GeForce GTX 1060 has 1280 cores and each of them is capable of running 16 threads; a similarly prices CPU—the Intel i7-7700—has only 4 cores each running 8 threads. It is worth mentioning that the processing cores on a GPU are not as sophisticated as a full CPU and run at a lower clock frequency. The processor clock frequency of the GTX 1060 is 1708 MHz whereas the i7-7700 has a base processor frequency of 3600 MHz. This means that GPGPU is more useful for applications that involve simple, repetitive, parallel tasks. A recent example is deep learning, where computations that follow the same logical sequence of control need to be performed on a deep network of nodes. Typical deep learning networks in 2015 consist of about one million nodes [6], which means that the computations cannot be done efficiently on a CPU.

This report investigates the gain in calculation speeds from the use of GPUs, and presents a diagnostic tool developed based on GPGPU, which can perform real-time histogram equalisation and FFT on images captured by an SEM. The tool was used to implement an automatic focusing and astigmatism correction algorithm, and the results are discussed.

# Chapter 2

# The gain in calculation speeds from the use of GPUs

## 2.1 GPU computing

The GPU is designed to meet the high demand in parallelism for fast rendering of scenes on displays. For example, a 1080p display refreshing at 60 Hz requires $60 \times 1920 \times 1080 = 124,416,000$ values to be computed in each second, which cannot be done efficient enough in the sequential manner used by the CPU. It describes an image using graphics primitives as shown in Fig. 2.1. To construct the image, the primitives are passed through the graphics pipeline of the GPU, which can be divided into the following stages:

- Vertex generation. A list of vertices are generated to represent the image as a 3D triangle mesh, as illustrated by Fig. 2.2.
- Triangle generation. The vertices are assembled into triangles, which are the fundamental hardware-supported primitive in modern GPUs.
- Fragment generation. The triangles are mapped to blocks of pixels on the screen; each block is called a "fragment".
- Fragment processing. The fragments are shaded based on colour and texture information to determine their final colour.
- Composition. A final image is created by assembling the fragments.

The processing of primitives within each stage follow the same logical sequence of control, but the data are different. This pattern is called "single instruction multiple data" (SIMD). The GPU has a large array of SIMD, multi-threaded processing cores sharing the same global memory, and it divides the cores among the stages such that the pipeline is divided in *space*, not time. Each primitive is processed by a thread of one of the processors.

The parallel structure of the GPU can also provide a significant speed improvement in general-purpose computing. Take the addition of two vectors of size $N$ as an example:

$$\boldsymbol{v_3} = \boldsymbol{v_1} + \boldsymbol{v_2}$$

A single-threaded CPU divides the task in

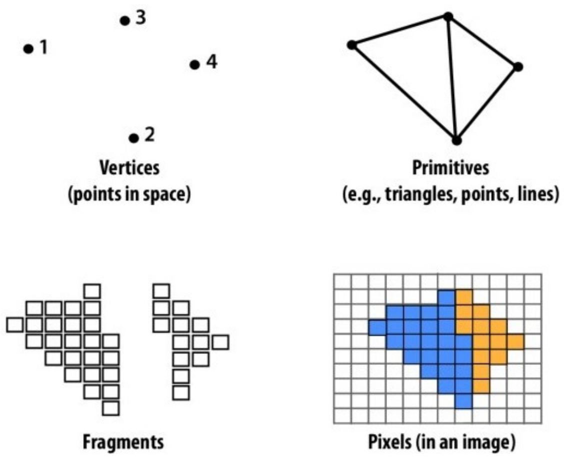


Figure 2.1: Graphics primitives [7].

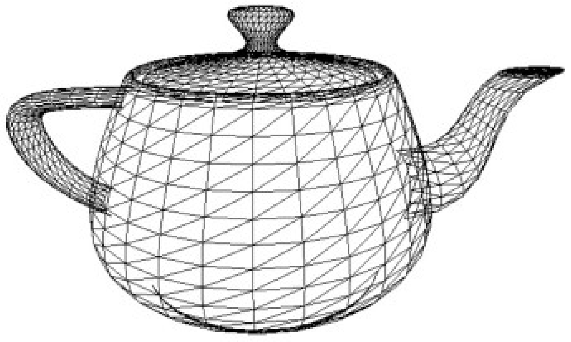


Figure 2.2: Representing an image as a 3D triangle mesh [7].

time and does:

$$\begin{aligned}&\text{Time 1: } \boldsymbol{v_3}[0] = \boldsymbol{v_1}[0] + \boldsymbol{v_2}[0]\\ &\vdots\\ &\text{Time n: } \boldsymbol{v_3}[n] = \boldsymbol{v_1}[n] + \boldsymbol{v_2}[n]\end{aligned}$$

This results in a time complexity of $O(N)$. The GPU can in theory achieve a time complexity of $O(1)$ by dividing the task in space, i.e. among threads, and doing:

$$\text{Time 1: } \begin{cases}\text{Thread 1: } \boldsymbol{v_3}[0] = \boldsymbol{v_1}[0] + \boldsymbol{v_2}[0]\\ \vdots\\ \text{Thread n: } \boldsymbol{v_3}[n] = \boldsymbol{v_1}[n] + \boldsymbol{v_2}[n]\end{cases}$$

The task is an SIMD task—the program takes one element from each of the vectors and perform addition on them, but the data handled by each thread is different, it can therefore make full use of the SIMD cores of the GPU. When $N$ is small, the overhead in allocating the resources means that the speed improvement is negligible; as $N$ grows larger, the reduction in time complexity quickly compensates for the overhead and makes the GPU much faster than the CPU; however, when $N$ is too big, the GPU will run out of resources, which sets a cap to its performance.

Prior to 2007, to use GPUs for general-purpose computing, the user must write programs using the graphics application programming interface (API) since it was the only interface to GPU hardware. The programming model can be summarised as below:

- The user specifies geometry that covers a region on the screen.
- The user sets parameters of the pipeline (e.g., “lighting” and “texture” information).
- The user provides the fragment processing program (kernel).
- The GPU produces an output “image” and stores it in global memory.

This was a major problem in GPGPU because many general-purpose tasks have nothing to do with graphics and are difficult to implement using the graphics API.

The introduction of CUDA changed the situation by providing a more natural, direct, non-graphics interface. The new programming model can be summarised as below:

- The user defines the computation as a structured grid of threads.
- The GPU executes each thread and stores the results in global memory.

This model allows the user to directly define threads that are run on the processing cores of the GPU, eliminating the complexity in translating the program into graphics pipeline language, which makes it easier for the user to take full advantage of the GPU’s power. The following section describes an experiment conducted to determine the gain in calculation speeds from using CUDA.

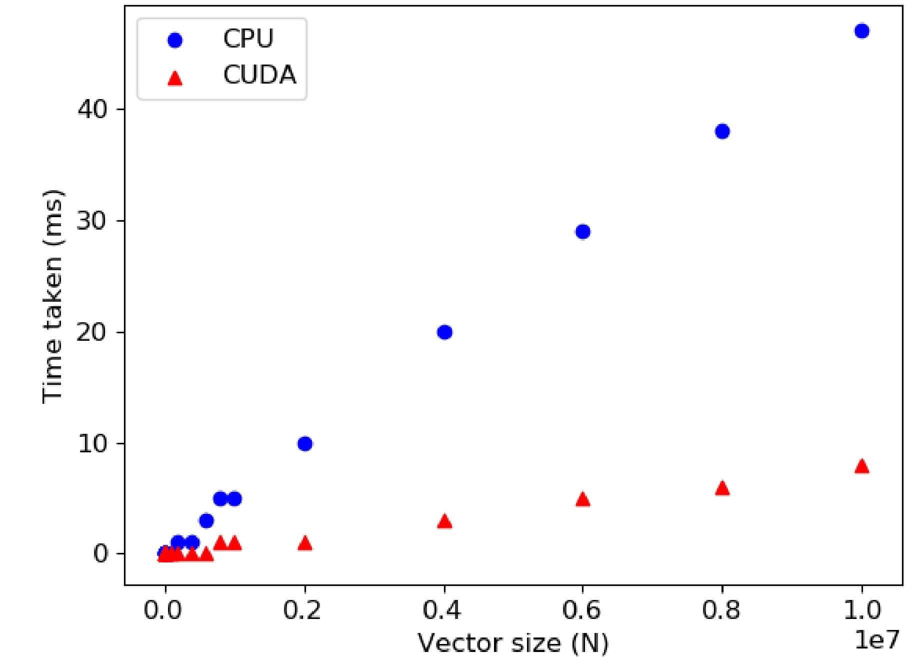
## 2.2 Experiment, results and discussions

An experiment was set up to compare the performance of a middle-range GPU—the NVIDIA GeForce GTX 1060—and a similarly priced CPU—the Intel Core i7-7700, for the following operations:

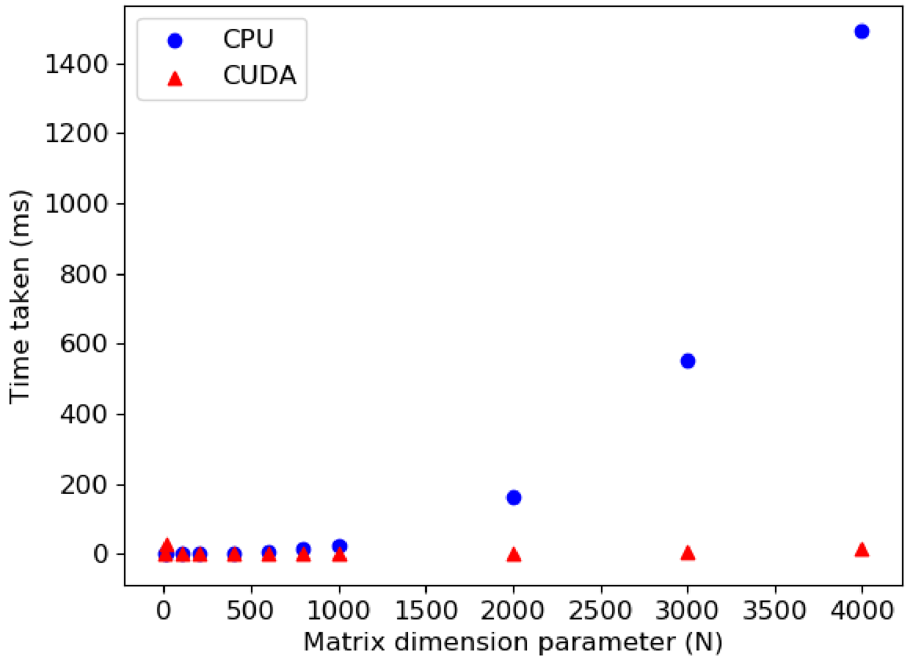
- Addition of two random vectors of size $N$, which has a time complexity of $O(N)$ if done without parallelism, as described in the previous section.
- Multiplication of two random matrices of dimension $N \times N$, which has a time complexity of $O(N^3)$ if done without parallelism (and without using special algorithms such as the Coppersmith-Winograd algorithm).

Ten test samples were taken for each set of parameters and the average results are shown in Fig. 2.3. As can be seen, the GPU provides a significant improvement on calculation speeds when there are a few millions of elements to be processed (which is typical in image processing).

This has allowed the development of a real-time diagnostic tool for the SEM with useful framerates, which is discussed in the next chapter.



(a) Vector addition.



(b) Matrix multiplication.

Figure 2.3: Performance test results of the GPU and CPU

# Chapter 3

# A real-time diagnostic tool for SEMs

## 3.1 The Design

**Features** The aim of the tool is to support interactive real-time diagnosis of SEM images that assists the operator or automated procedures, and it has two main features:

- Real-time histogram calculation and histogram equalisation.
- Real-time FFT calculation.

**Design principle** Considering that the tool may serve as the foundation stone for many further developments, the design of the software has a strong emphasis on readability and maintainability.

**Selection of programming language** The SEM used for the project is made by Carl Zeiss AG, which provides an API for controlling the SEM and grabbing images from it. The API supports C++ and thus makes it a possible programming language for the project. Being a relatively low-level compiled language makes C++ extremely fast and useful for speed-critical applications. However, it also means that C++ has a complex syntax, which could significantly slow down development if the user does not have enough experience with it. Considering the time scale of the project and to make development easier for people who will continue the work, Python is selected instead of C++ as the programming language. It has a much simpler syntax and is widely supported; with careful design, it has proved to produce useful results despite its slower speed (see later sections).

**Modules of the software** The software is highly modularised for maximising readability and maintainability. This also makes it easier to translate the program into C++ later if faster speed is needed since C++ is mostly used in an object-oriented manner. The software is divided into six main classes as shown in Fig. 3.1:

- *SEM_API* is a Python wrapper for the native SEM API, which allows the user to directly control the SEM in Python.
- *SemImage* encapsulates variables and methods that are directly relevant to an SEM image.

- *SemImageGrabber* is a helper class that helps obtain image data from the SEM and create instances of *SemImage* from them. Images can be obtained in two ways:

  - From the SEM.
  - From a local folder, which is helpful when the SEM is not available.

  *SemImageGrabber* detects if the SEM is available and decides on which way to use.

- *SemTool* handles the creation and rendering of the graphical user interface (GUI). Fig. 3.2 shows a screenshot of the control panel. The pushbuttons open a window for the corresponding plot and the radio buttons select algorithms to be performed on the image [see section TBC]. The plots are shown on different windows and can be opened and closed individually. This improves framerate by allowing the user to close unneeded windows, and also makes it easier to add other plots later. Table 3.1 shows that the improvement is significant, as rendering windows is a major time-consuming process of the software. When only one window is opened, a refresh rate of about 16 frames per second is possible. There are two ways for updating the plots, and the first one is to re-render the whole window. This wastes time since some components are always the same and do not need to be re-rendered, such as the window title and the axes. Therefore, *SemTool* uses the second method, where only the data to be plotted are updated. This directly modifies the relevant data in the memory of the GPU, thereby avoiding re-rendering the whole window. Tests have shown that if the first method is used, the time taken to update each frame when there is only one plot opened will be 90 ms instead of 60 ms.

- *SemCorrector* implements the automatic focusing and astigmatism correction algorithm and uses a helper class *Masker* to achieve fast computation [see section TBC]. Due to the impact of the COVID-19 pandemic, the algorithm has not been fully tested and is therefore not integrated to the GUI yet.

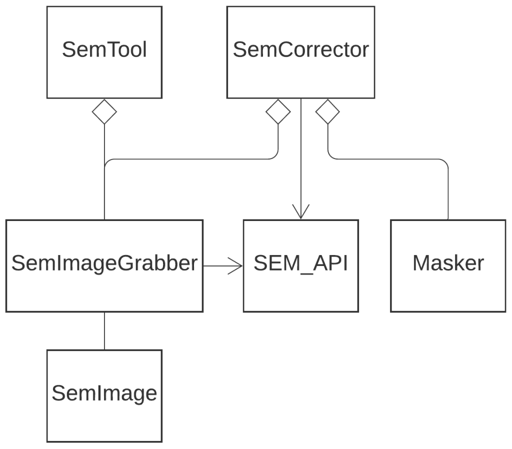


Figure 3.1: Class diagram of the software in unified modelling language (UML).

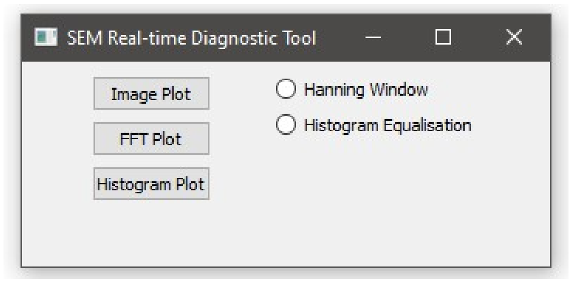


Figure 3.2: Screenshot of the GUI.

Table 3.1: Results of software framerate test

| **Number of plots opened** | **Time taken to update each frame** |
|---|---|
| zero | 15 ms |
| one | 60 ms |
| three | 160 ms |

[a]The numbers are approximate.
[b]8-bit grey-scale 1024 × 768 input images.
[c]On Intel Core i7 and NVIDIA GTX 1060

## 3.2 Real-time histogram calculation and histogram equalisation

Histograms use bars to represent the shades of tones (level) that make up an image. The grey level (brightness level) histogram is the most relevant histogram to SEM images, since SEMs translate the energy of the secondary electrons directly into a grey level. The height of the bars in a grey level histogram can be calculated by

$$n_l = \frac{1}{P}\sum_p I(l_p = l), \tag{3.1}$$

where $n_l$ is the normalised number of pixels of grey level $l$, $P$ is the total number of pixels in the image and $l_p$ is the grey level of the $p_{th}$ pixel.

Fig. 3.3a shows an 8-bit grey-scale image, i.e. its depth of digitisation is 8-bit and it has 256 grey levels. Fig. 3.3b shows the histogram of the image and its integral. As can be seen in the histogram, most pixels are concentrated in the middle of the greyscale. This is reflected by the fact that the image is missing its highlights and shadows.

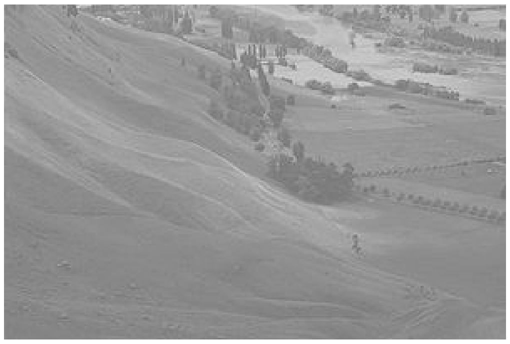
Histogram equalisation is a method for adjusting the distribution of pixel intensities of an image, to improve its overall contrast. Effectively, it is achieved by spreading out more frequent intensity values. To perform histogram equalisation, firstly obtain the integral of the histogram using

$$s_l = \sum_{i=1}^{l} n_i,$$

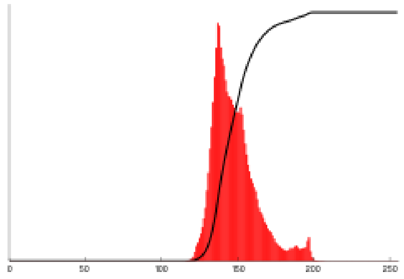
where $s_l$ is the value of the integral at grey level $l$. The integral spans from 0 to 1 as the histogram is normalised. Scale the integral by the maximum grey level and perform rounding to create the transform function

$$f_l = \lfloor s_l L \rceil, \tag{3.2}$$

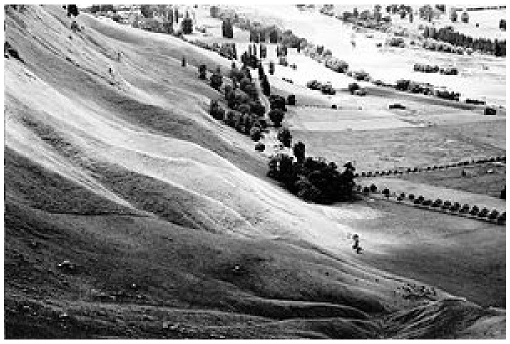
If a pixel in the original image has a grey level $l$, it will have a grey level $f_l$ in the transformed image. For example, if all pixels in the original image are concentrated between grey level 100 to 200, the pixels of level 100 will become 0 in the new image while the pixels of level



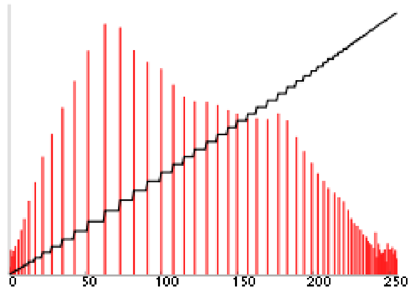
(a) Original image.



(b) Histogram of the original image.



(c) Histogram-equalised image.



(d) Histogram of the histogram-equalised image.

Figure 3.3: Histogram and histogram equalisation.

200 will become 255 (assuming an 8-bit image).

The result is more noticeable when the image has low contrast, such as the one shown in Fig. 3.3a. The histogram-equalised version of it is given in Fig. 3.3c and the new histogram in Fig. 3.3d. More details are now visible as the contrast has been enhanced.

The most time-consuming part of histogram equalisation is to map all pixels in the original image to their new values using the transform function (3.2). This can be implemented using CPU code in one line:

```
newImage =
  map(lambda x: f[x], image)
```

`f` is an array that serves as the transformation function, if the original pixel value is `x`, the new value should be `f[x]`. The `lambda` operator is a way to create small anonymous functions, which are throw-away and are only needed where they are created. It improves conciseness and readability by reducing code bloat. `lambda x: f[x]` creates an anonymous function that returns `f[x]` when given `x`, and `map()` uses this function to transform all pixels in `image`. This gives a time complexity of $O(N^2)$ which is not optimal. A smaller time complexity can be achieved using the GPU code:

```
map = cupy.ElementwiseKernel(
  'T x, raw T f', 'T xNew',
  'xNew = f[x]',
  'map'
)
```

```
newImage = map(image, f)
```

`cupy.ElementwiseKernel()` defines a kernel that the GPU uses to process the input data in parallel. Tests have shown that the use of GPU can increase the speed by 2 times, as summarised in Table 3.2. In theory, the factor of speed improvement should be similar as that for calculating the histogram, since it also requires iterating through all pixels in the image. However, the code implementations introduce overheads which limit the improvement in speed.

Table 3.2: Speed comparison between CPU and GPU

| **Algorithm** | **CPU** | **GPU** |
|---|---|---|
| Apply Hanning window | 16 ms | 9 ms |
| Calculating histogram | 30 ms | 2 ms |
| Histogram equalisation | 9 ms | 5 ms |
| Fast Fourier transform | 60 ms | 8 ms |

[a]The results are averaged over 10 samples.
[b]8-bit greyscale 1024 × 768 input images.
[c]On Intel Core i7 and NVIDIA GTX 1060.

## 3.3 Real-time fast Fourier transform calculation

# Chapter 4

# Automatic focusing and astigmatism correction algorithm

# Chapter 5

# Conclusions

# Bibliography

[1] C. REEVES, “The uses of scanning electron microscopy for studying semiconductor devices,” International Journal of Electronics, vol. 77, no. 6, pp. 919-928, 1994, doi: 10.1080/00207219408926111.

[2] T. Cushnie, N. O’Driscoll and A. Lamb, “Morphological and ultrastructural changes in bacterial cells as an indicator of antibacterial mechanism of action,” Cellular and Molecular Life Sciences, vol. 73, no. 23, pp. 4471-4492, 2016, doi: 10.1007/s00018-016-2302-2.

[3] “SEM A to Z,” Jeol.co.jp, [Online], available: https://www.jeol.co.jp/en/applications/pdf/sm/sem_atoz_all.pdf. [Accessed: 18 May 2020].

[4] A. Vladár, M. Postek and M. Davidson, “Image sharpness measurement in scanning electron microscopy-part II,” Scanning, vol. 20, no. 1, pp. 24-34, 2006, doi: 10.1002/sca.1998.4950200104.

[5] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, “GPU Computing,” in Proceedings of the IEEE, vol. 96, no. 5, pp. 879-899, May 2008, doi: 10.1109/JPROC.2008.917757.

[6] Ian Goodfellow; Yoshua Bengio; Aaron Courville, “Deep learning,” in Deep Learning, The MIT Press, 2017, p. 23.

[7] “GPU Architecture and CUDA Programming,” Carnegie Mellon University, 2017, [Online], available: http://15418.courses.cs.cmu.edu/spring2017/home. [Accessed: 19 May 2020].