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

Liuchuyao Xu

May 26, 2020

Contents

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 difficult or impossible to observe. For example, SEMs are an essential quality control tool for wafer manufacturing processes in the semiconductor industry [1]. A review SEM detects the defects on a die by comparing an image of it with a reference, and then stores the image for further use. The inspection would have been impossible without the aid of SEMs.

Fig. 1.1 shows the basic construction of the SEM. 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 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 are emitted from the specimen. They are called secondary electrons and are collected by the detector, which generates signals based on their energy levels. The display unit converts the signals into images; one image is produced after each complete scan of the specimen.
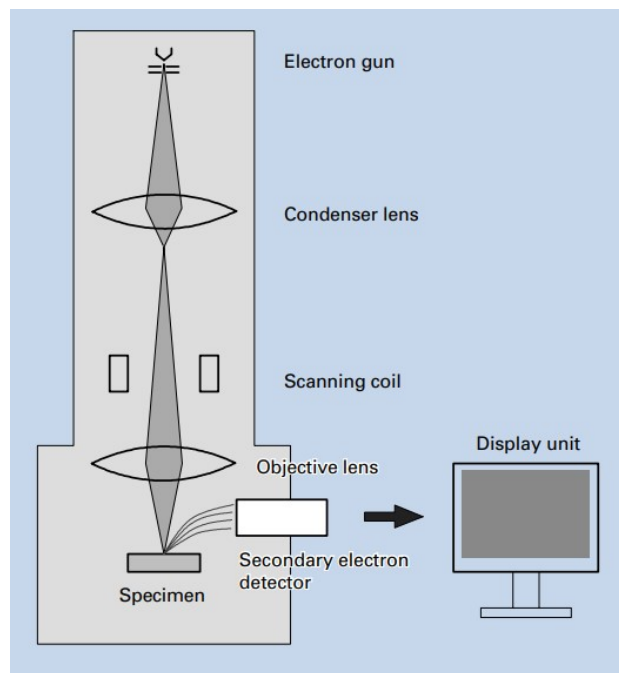

Figure 1.1: Basic construction of the SEM [2].

Many image analysis methods are used on SEM images. The fast Fourier transform is an especially important one since it can be used not only to study the structure of the specimen, but also to evaluate the focusing and astigmatism of the images [see section 3.3.2]; 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 [3]. 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 was originally designed to be a highly specialised hardware that excels in rendering complex, high-resolution, real-time 3D scenes for games; however, its special architecture has allowed it to outperform CPUs in many other areas and resulted in the birth of the idea of general-purpose computing on graphics processing units (GPGPU) [4], where GPUs are used to perform computations that are traditionally handled by CPUs.

A key characteristic of GPUs that differentiate them from CPUs is massive parallelism. Different pixels in a 3D scene require different processing to achieve visual effects such as lighting, blurring, and fogging. GPUs do this by breaking down the scene into fragments and manipulating each fragment independently. Modern GPUs have thousands of parallel processor cores each running tens of parallel threads to meet the requirement. 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 Core i7-7700HQ—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 2800 MHz and a maximum of 3800 MHz. This means that GPGPU is more useful for applications that involve simple, repetitive, parallel tasks. A recent example is deep learning, where the same computation needs to be performed on a large network of nodes. Typical deep learning networks in 2015 consist of about one million nodes [5], which makes the computations impossible to be performed efficiently by CPUs.

This report studies the gain in computation speeds that can be obtained by using GPUs, and presents a real-time diagnostic tool made possible by that. The tool assists SEM operators by providing the histogram and FFT of the images in real-time. It was used to implement an automatic focusing and astigmatism correction algorithm for SEMs, and the performance is discussed.

Chapter 2

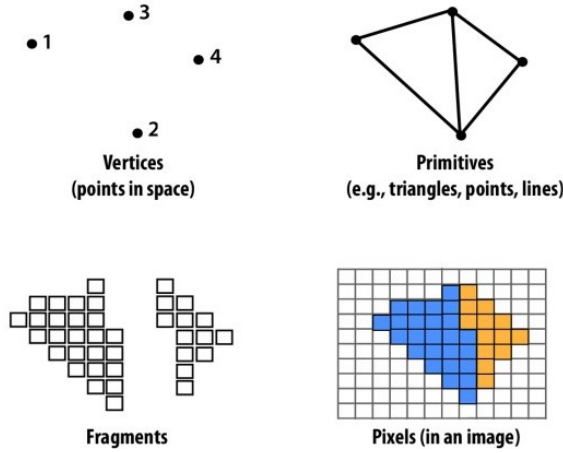# The gain in computation speeds from the use of GPUs

2.1 GPU computing

GPUs are super parallel work distributors; they describe an image using a set of graphics primitives and allocate resources to manipulate each of them independently. Fig. 2.1a shows the primitives used by modern GPUs. Different types of primitives are created and processed in different stages of the graphics pipeline, as described below:
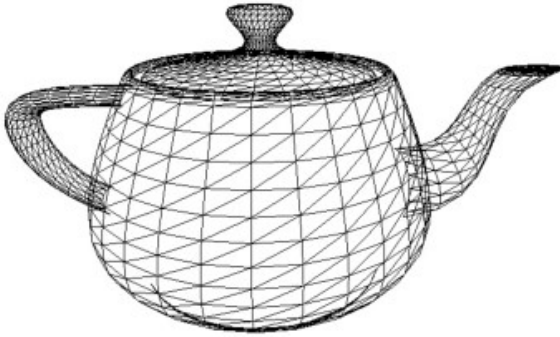
- Vertex fetching. The GPU fetches a list of vertices that represent the image as a 3D triangle mesh, as illustrated in Fig. 2.1b. Triangles are the fundamental hardware-supported primitive in modern GPUs; their advantage is that any of them can be defined by three vertices, which simplifies maths. Vertices are generated outside the GPU, together with information on lighting, texture, and transforms that need to be applied.

- Vertex shading. Mathematical operations are performed on the vertex data to add desired effects.

- Rasterisation. Each triangle is mapped to a block of pixels on the screen, called a "fragment".

- Fragment shading. The fragments are shaded on a per-pixel basis to determine their final colour.

- Raster operations. A final image is constructed by assembling the fragments and put into the frame buffer.

The processing units of the GPU support the data parallelism by adopting a single-program multiple-data (SPMD) programming model, where the same shading and rasterisation programs are used to to process different data. The programs can read data from shared global memory and write back to it. The GPU also makes use of task parallelism by dividing the pipe in *space*, not time. Some pixels of a triangle could be running fragment-shading programs while others are already being written to the frame buffer; simultaneously, the GPU could be fetching the vertices of another triangle.

The parallelism can also provide a performance improvement for non-graphics tasks. 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 time and does:

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

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 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 vector addition is an SPMD task—each thread takes one element from each of the vectors and performs addition on them, but different threads handle different data. When N is small, the overhead in allocating the threads 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, the GPU will run out of resources when N is too big, and this sets a cap to its performance.

(a) Graphics primitives.

(b) An image can be represented as a 3D triangle mesh.

Figure 2.1: The GPU describes an image as a set of primitives [6].

The vertex and fragment shading programs used to be fixed and graphics-specific, and the progression in making them customisable is a key that pushed the advancement of GPGPU [4]. Prior to 2007, to use GPUs for general-purpose computing, the user must write programs using the language of the graphics application programming interface (API) since it was the only interface to GPU hardware:

- The user specifies a list of "vertices" that cover a region on the screen.

- The user sets parameters of the graphics pipeline, e.g. "lighting" and "texture".

- The user provides "shading" programs for the vertices and fragments.

- The GPU produces an output "image" and stores it in global memory.

This was a major problem in GPGPU because many tasks have nothing to do with graphics and are difficult to translate into the graphics language. 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 puts the result in global memory.

This eliminates the complexity in using the graphics API and allows the user to directly define threads that are run on the processing cores of the GPU.

## 2.2 Experiment, results and discussions

An experiment was conducted to determine the gain in computation speeds that can be obtained by using CUDA. A middle-range GPU—the NVIDIA GeForce GTX 1060—and a similarly priced CPU—the Intel Core i7-7700HQ—are used to perform addition of two random vectors of size N, which has a time complexity of $O(N)$ if done without parallelism.

Fig. 2.2 gives the results of the test. Measuring the gradient of the best-fit lines shows that the GPU is 12000 times faster, which agrees with the fact that the GPU is capable of running 20480 parallel threads while the CPU was only allowed to use one. It is possible to improve the performance of the CPU by making it use all of its 32 parallel threads, but it will still be hundreds of times slower. The maximum resolution of the GTX 1060 is 7680 × 4320 at 60 Hz, which means it can calculate two billion values within each second; if the Core i7 is used, that will take 200 minutes with one thread or 6 minutes with 32 threads.

Fig. 2.2b shows that the computation time of the GPU does not scale linearly with the size of the vectors, but increases with steps. This confirms that the GPU can achieve time complexity of $O(1)$ within a limit, as described in the previous section.
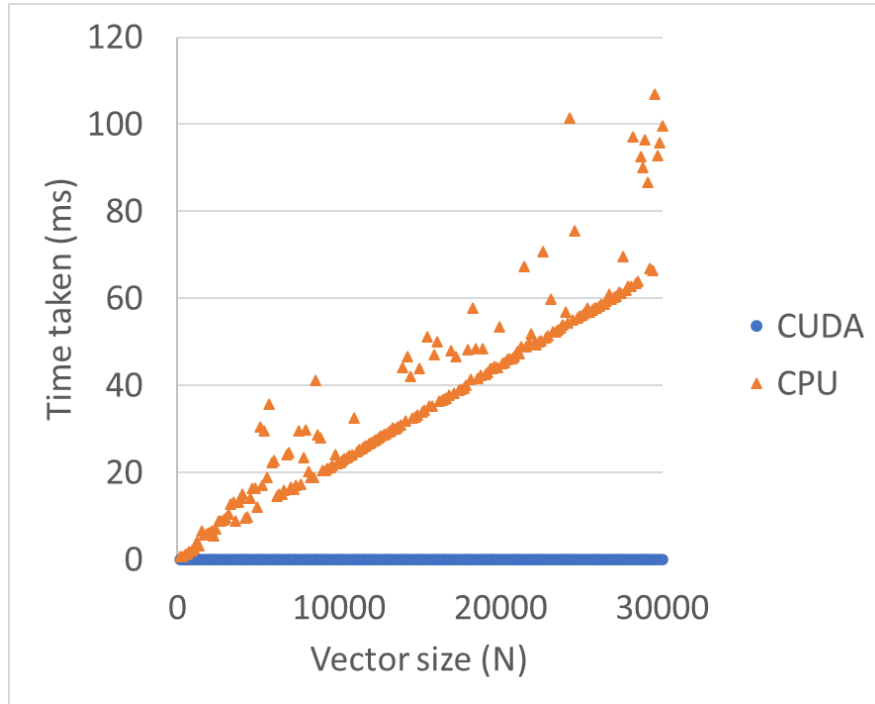
(a) Results of both the GPU and the CPU.
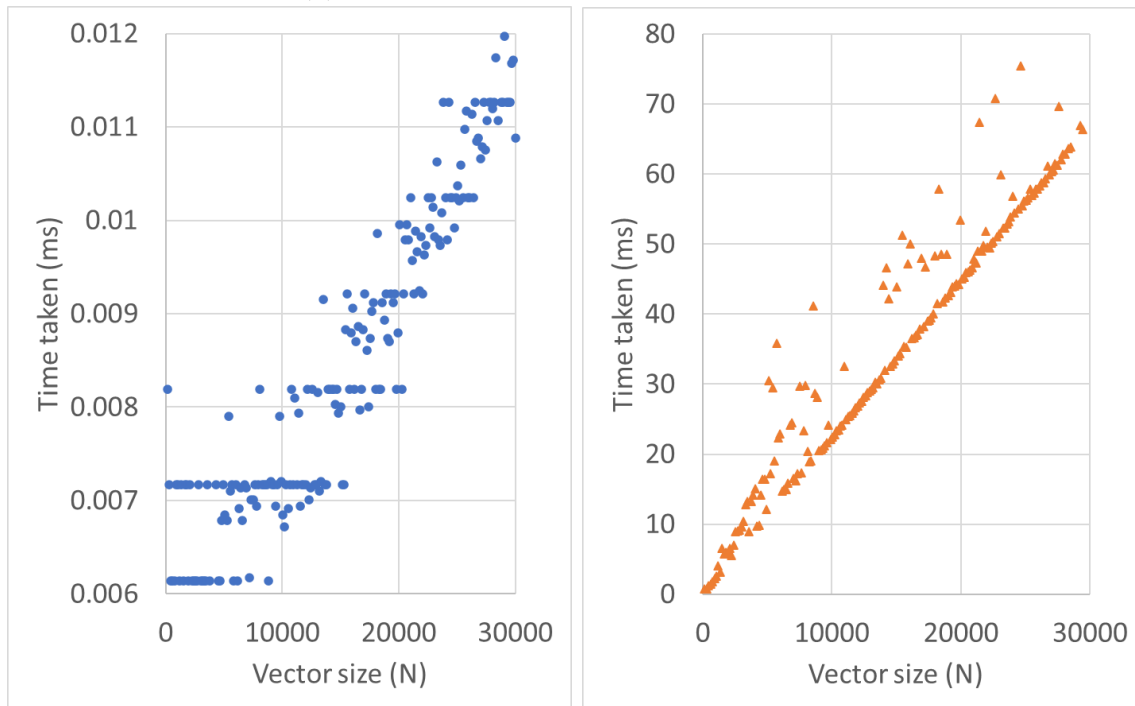
(b) Results of the GPU.

(c) Results of the CPU.

Figure 2.2: Performance on vector addition of the GPU and the CPU

Chapter 3

A real-time diagnostic tool for SEMs

3.1 Design of the software

A diagnostic tool was developed based on fast computations provided by CUDA. The aim of the tool is to support interactive real-time diagnosis of SEM images that assists operators or automated procedures, and it has two main features:

- Real-time histogram computation.

- Real-time FFT computation.

Considering that the tool may serve as the foundation stone for many further projects, the design of the software has a strong emphasis on readability and maintainability. This has affected the selection of programming language and the architecture of the software.

NVIDIA developed CUDA and provides an API for programmers to use it. The API is written in C and thus makes it a candidate programming language for developing the tool. Being a relatively low-level compiled language makes C extremely fast and useful for speed-critical applications. However, it also means that it has a complex syntax, which could significantly slow down development if the user does not have enough experience with it. Taking into account the time scale of the project and to make development easier for people who will continue the work, Python was used instead of C. It has a much simpler syntax and is widely supported; with careful design, it has proved to be able to produce useful results despite its slower speed.

To maximise quality, the software is highly modularised containing the following main classes:

- *SEM_API*. This is a Python wrapper for the native SEM API written by Luyang Han, which allows the user to directly control the SEM and grab images from it in Python.

- *SemImage*. This class encapsulates variables and methods that are directly relevant to SEM images. For example, it provides a function for applying a Hann window on the image, and a function for calculating the FFT of the image.

- *SemImageGrabber.* This class provides a function for grabbing images from the SEM and creating instances of *SemImage* from them. When an SEM can be detected, images are obtained by reading the memory of the SEM; otherwise, they are taken from a local folder specified by the user.

- *SemTool.* This class handles the rendering of the graphical user interface (GUI); it is the entry point of the software. The user uses the GUI to specify which algorithms to be performed on the SEM images and what diagnostic information to be displayed.

- *SemCorrector.* This class implements an automatic focusing and astigmatism correction algorithm for SEMs; it uses the *Masker* class to ensure fast computing [see chapter 4].

Fig. 3.1 shows the class diagram of the software. Dividing the software into multiple classes has many benefits, including the following:

- The encapsulation of data and methods makes it easier to keep track of them, which improves readability. For example, *SemImage* can be used as below:

```
image = semImageGrabber ()
image.applyHanning ()
image.updateFft ()
image.updateHistogram ()
```

- It splits a big task into smaller ones, and thus improves maintainability by allowing the implementation of different tasks to be modified without changing other tasks. For example, *SemImageGrabber* is implemented using 20 lines of code and used at four different places within *SemCorrector*, by one line of code:

```
image = semImageGrabber ()
```

Without *SemImageGrabber*, *SemCorrector* will require 76 more lines of code which all have to be changed whenever the programmer wants to implement a new method for grabbing images.

- Classes can be created by inheriting from other classes. This improves reusability by allowing subclasses to hold all the data and perform all the actions of their superclasses, and thus speeds up further developments.
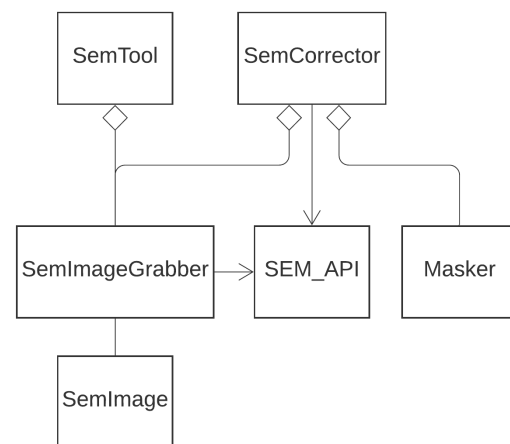

Figure 3.1: Class diagram of the software in unified modelling language (UML).

3.2 Real-time histogram computation

3.2.1 Theory of the histogram

Histograms use bars to graphically represent the distribution of tones in an image; the heights of the bars give an idea of the strength of the shadows, exposure, and highlights of the image [7]. Photographers use them to determine whether a photo has been under-exposed or over-exposed.

HSB—hue, saturation, and brightness are one set of tones commonly used today, and the brightness level (also called grey level) is the most relevant to SEM images. This is because SEMs translate the energy of the secondary electrons directly into a grey level. The grey level histogram of an image can be calculated using

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

where P is the total number of pixels in the image and l_p is the grey level of the pth pixel. n_l is the height of the bar representing grey level l, and also the normalised number of pixels of grey level l.

Fig. 3.2a shows an 8-bit greyscale image, i.e. it has 256 possible grey levels. Fig. 3.2b shows the histogram of the image and its integral. The image is missing its shadows and highlights, which is reflected by a histogram with most pixels concentrated towards the middle.
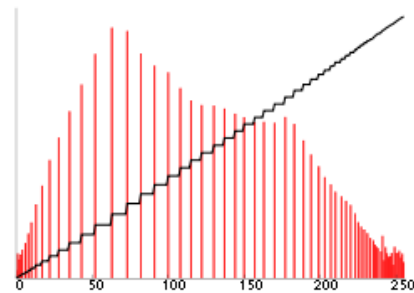

(a) Original image.
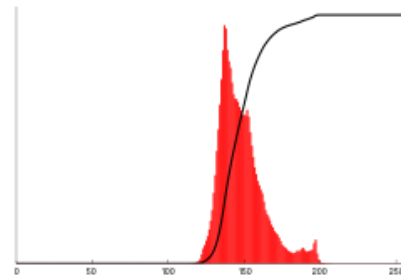
(b) Histogram of the original image.

(c) Histogram-equalised image.

(d) Histogram of the new image.

Figure 3.2: Histogram and histogram equalisation.

3.2.2 How the histogram can aid SEM operators

The histogram allows SEM operators to compare images in terms of their contrast and intensity. Because image intensity is directly related to the number of electrons collected by the SEM, and thus the structure of the surface under observation, the histogram provides an alternative way for studying specimen structures. For example, the appearance of uneven steps or thin protrusions on the specimen surface may be affected by edge effects [2], which make their edges look like thick bright lines instead of sharp lines. This results in strong highlights in the histogram of an otherwise darker image.

The histogram can also be used to enhance the contrast of SEM images via histogram equalisation, which is an algorithm for adjusting the distribution of tones of an image. Effectively, it is achieved by spreading out the more frequent tonal 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 since the histogram is normalised. Scale the integral by the maximum grey level L and perform rounding to create the transform function

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

If a pixel in the original image has grey level l, it will have 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 200 will become 255 (assuming an 8-bit image). The use of the integral ensures that more concentrated pixels are spread out more, because they result in higher values of s_l and are thus mapped to a wider new range by the transform function.

Improved contrast brings out details of the image, as illustrated in Fig. 3.2c. This can help SEM operators identify small features that are hidden by a lack of contrast. The tool supports histogram equalisation in real-time.

The importance of real-time computation is that it allows the operator to see the histogram and the enhanced image while constantly adjusting settings of the SEM, which boosts productivity.

3.2.3 Some details on implementations (a demonstration of GPU programming)

The most time-consuming part in performing histogram equalisation is to map all pixels in the original image to their new values using the transform function (3.2). It can be implemented on the CPU using the following

Python code, and has a time complexity of $O(N^2)$ if run using one thread only:

```
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`.

A smaller time complexity can be achieved by using CUDA, and the Python implementation will look like:

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

`cupy.ElementwiseKernel()` defines a kernel function that is used by the parallel processing cores of the GPU to process the input data.

C is needed to use the native CUDA API, and the implementation will look like:

```
map(int *image,
    int *f,
    int *newImage,
```

```
    int numElements)
{
    int i = blockDim.x
            * blockIdx.x
            + threadIdx.x;

    if (i < numElements)
    {
        newImage[i] =
                f[image[i]]
    }
}
```

```
int i = blockDim.x
    * blockIdx.x
    + threadIdx.x;
```

is the key for SPMD in the CUDA programming model. CUDA groups the available threads into a matrix of blocks. All threads run the same `map` function, and this line produces a different index number within each thread using the block dimension, block ID, and thread ID. The threads use their index numbers to manipulate data at different locations in the memory.

3.3 Real-time FFT computation

3.3.1 Theory of the FFT

The Fourier transform (FT) decomposes a function into its constituent frequencies, as illustrated in Fig. 3.3. The composite wave can be represented by the superposition of a 1 Hz sine wave and a 10 Hz sine wave, and its

FT therefore has two components. A function that changes more abruptly will have stronger high-frequency components than a smoother one.
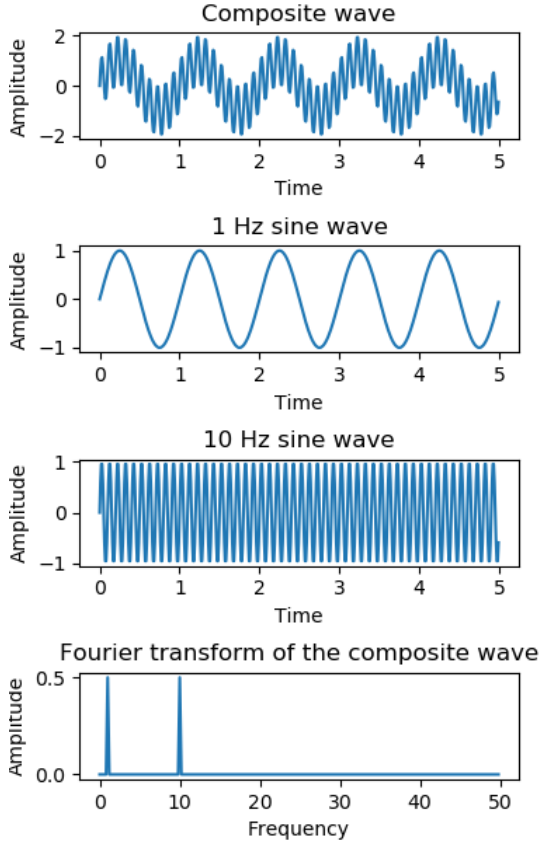

Figure 3.3: Fourier transform

In image processing, the discrete Fourier transform (DFT) is used instead of the FT, which takes a finite sequence of equally-spaced values as input and is thus better suited than the FT. It is defined by

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}, \quad 0 \le k \le N-1,$$

(3.3)

where $x_0, x_1, ..., x_{N-1}$ is the input sequence and $X_0, X_1, ..., X_{N-1}$ is the output sequence. X_k is effectively the result of the dot product between the vector $[x_0, x_1, ..., x_{N-1}]$ and $[e^0, e^{-i2\pi k/N}, ...,, e^{-i2\pi k(N-1)/N}]$, and the latter is a finite sequence of frequency $2\pi k$. Therefore, the DFT computes the magnitude of the component of the input sequence that has frequency $2\pi k$, for $0 \le k \le N-1$.

A drawback of using (3.3) is that it has a time complexity of $O(N^2)$ (N multiplications need to be computed for each of the N components in the output sequence $X_0, X_1, ..., X_{N-1}$). In image processing, where the inputs are 2D sequences, the time complexity quickly explodes and makes the equation practically impossible to use. A fast Fourier transform (FFT) is an algorithm that computes the DFT with smaller timer complexity. There are many feasible FFT algorithms and the most common one is the Cooley-Tukey algorithm [8]. The algorithm is used where N is, or can be chosen to be, a highly composite number. Suppose $N = r_1 \cdot r_2$, then the indices in (3.3) can be expressed as

$$n = n_1 r_2 + n_0, \quad n_0 = 0, 1, ..., r_2 - 1,$$
$$n_1 = 0, 1, ..., r_1 - 1.$$

Equation (3.3) can then be written as

$$X_k = \sum_{n_0} \sum_{n_1} x_{n_0+n_1} \cdot e^{-i2\pi k n_1 r_2/N} e^{-i2\pi k n_0/N}$$
$$= \sum_{n_0} e^{-i2\pi k n_0/N} \sum_{n_1} x_{n_0+n_1} \cdot e^{-i2\pi k n_1 r_2/N}.$$

Now, only $r_1 + r_2$ operations need to be performed to compute X_k, and the whole algorithm requires $N(r_1 + r_2)$ operations. The procedure can be repeated to produce an m-step algorithm requiring $N(r_1 + r_2 + \cdots + r_m)$ operations, where $N = r_1 \cdot r_2 \cdots r_m$. Modern processors can perform the algorithm the most efficiently when $r_1 = r_2 = \cdots = r_m = 2$ because of their binary architecture, then the total number of operations is

$$2N \log_2 N,$$

which yields a time complexity of $O(N \log N)$.

Fig. 3.4 gives an example of FFT applied to a real image. The top row shows a set of images and the bottom row shows the corresponding transform (amplitudes only). Points near the origin represent lower-frequency components and are cut off when a high-pass filter is applied to the image.
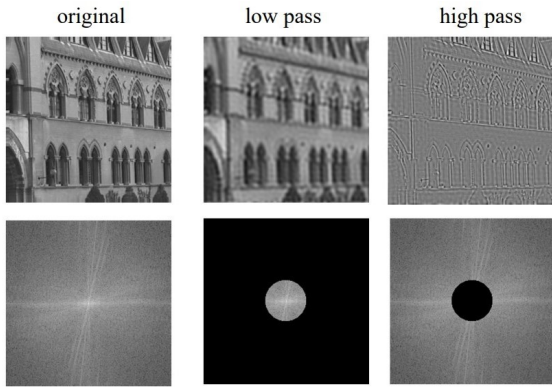

Figure 3.4: FFT and filtering on images [9].

A problem in the use of the FFT for image analysis is the boundary effect. The abrupt discontinuity at the edges gives the FFT some strong high-frequency components. In practice, windows functions such as the Hann window and the Hamming window are often applied to the image before performing the FFT, which taper the edges off towards zero and thus reduce the boundary effect. Fig. 3.5 shows how the Hann window and the Hamming window look like; they can be obtained by setting a_0 to 0.5 and $25/46$, respectively, in the function

$$w[n] = a_0 - (1 - a_0) \cdot \cos \frac{2\pi n}{N}, \quad 0 \le n \le N. \tag{3.4}$$
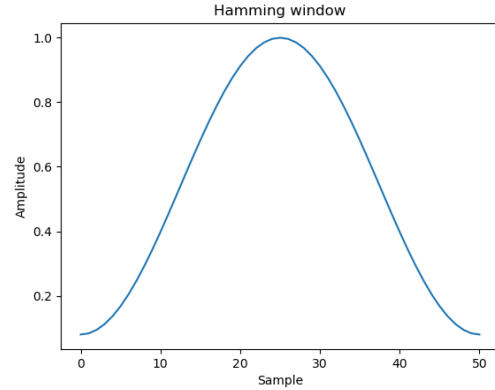

(a) Hamming window.
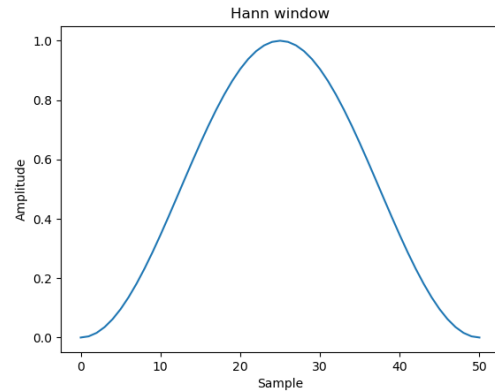
(b) Hanning window.

Figure 3.5: Window functions.

3.3.2 How the FFT can aid SEM operators

The quality of an SEM image is affected by aberrations. While some exist because of the fundamental properties of the microscope and are difficult to get rid of, some can be eliminated by adjusting relevant settings. Two important ones are focus and stigmator control, which directly affect the resolution and astigmatism of the image, respectively. Fig. 3.6 illustrates the effect of wrong focus and stigmator settings.
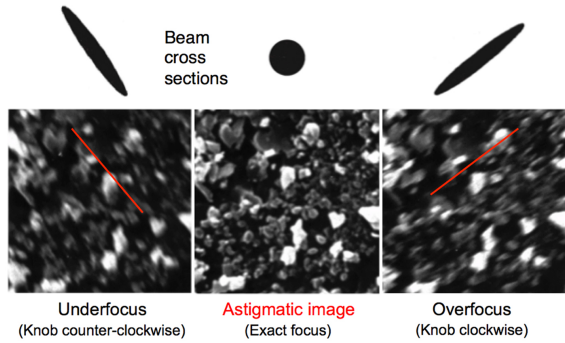
(a) Astigmatic images with different level of focusing.
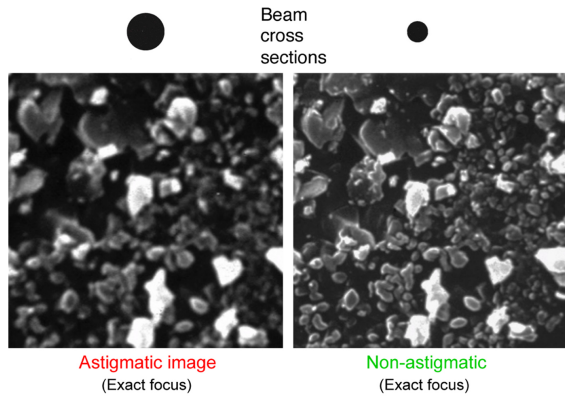
(b) Exact-focus images with different level of astigmatism.

Figure 3.6: Sample astigmatic SEM images [10].

Focus determines the focal point of the electron probe. When the focal point is far from the surface of the specimen, the incident electrons interact with the specimen in a larger area. As a result, spots near each other produce signals of closer magnitude. This makes the image appear blurry.

Stigmators are used to compensate for astigmatism. Astigmatism arises due to imperfections in components of the SEM and describes uneven focus in the electron probe, as illustrated in Fig. 3.7. When the electron probe is out of focus, astigmatism makes the incident electrons interact with the specimen in an elliptical area, and thus makes the image appear stretched. When the electron probe is in focus, astigmatism makes the image appear blurry.

Although experienced SEM operators can often tell the accuracy of focusing and level of astigmatism in a short time, it may not be as straightforward for new users. The complexity arises because any judgement of an image is based on what the operators see through their eyes, which is subjective. Sometimes, the surface being observed may have a complex structure and makes judgement even harder. Intensive training and practical experience are often required for an operator to become efficient in using the SEM.

The real-time FFT calculation aids the operators by providing an alternative way of evaluating the focusing and astigmatism of
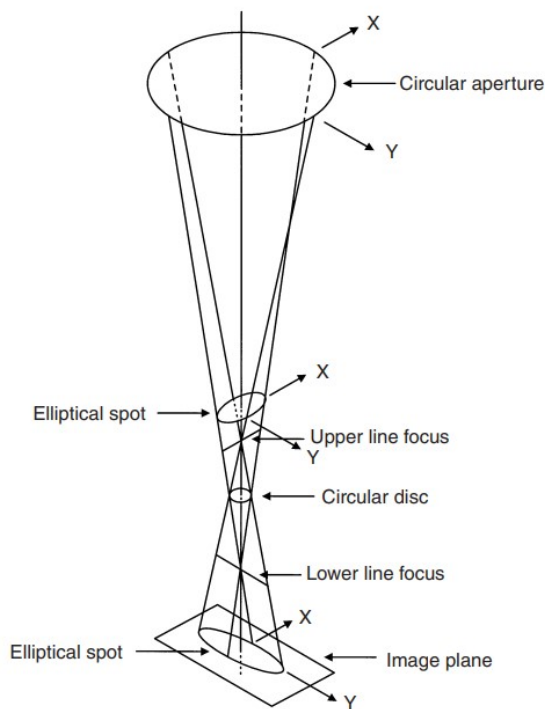
Figure 3.7: Uneven focus of the SEM.

3.4 Performance tests of the tool

Experiments were conducted to test the performance of the tool with the NVIDIA GeForce GTX 1060 and the Intel Core i7. The following algorithms were performed on a set of 8-bit 1024×768 SEM images and the average results are shown in table 3.1. If all the algorithms need to be performed on each image, using the CPU will limit the framerate to about 9 fps. The use of GPU changes the number to 42 fps, which gives margin to other processes such as the rendering of the GUI.

Table 3.1: Results of performance tests of the diagnostic tool

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

images using the FFT. Fig. 3.8 provides a set of sample images that illustrate how different degrees of defocus and astigmatism affect the FFT of an image. An in-focus image contains more details, and its FFT will thus have stronger high-frequency components. The FFT of an astigmatic image rotates by 90 degrees as the image goes from under-focus from over-focus; this is because the elliptical incidence area of the electron probe rotates by 90 degrees and makes the image appear stretched in the new direction (as shown in Fig. 3.6).

The importance of real-time calculation is that it saves SEM operators time by eliminating the need for downloading each image, and thus improves their productivity.

(a) Underfocus and no astigmatism   (b) In-focus and no astigmatism   (c) Overfocus and no astigmatism

(d) Underfocus and stigma x decreased   (e) In-focus and stigma x decreased   (f) Overfocus and stigma x decreased

(g) Underfocus and stigma x increased   (h) In-focus and stigma x increased   (i) Overfocus and stigma x increased

(j) Underfocus and stigma y decreased   (k) In-focus and stigma y decreased   (l) Overfocus and stigma y decreased

(m) Underfocus and stigma y increased   (n) In-focus and stigma y increased   (o) Overfocus and stigma y increased
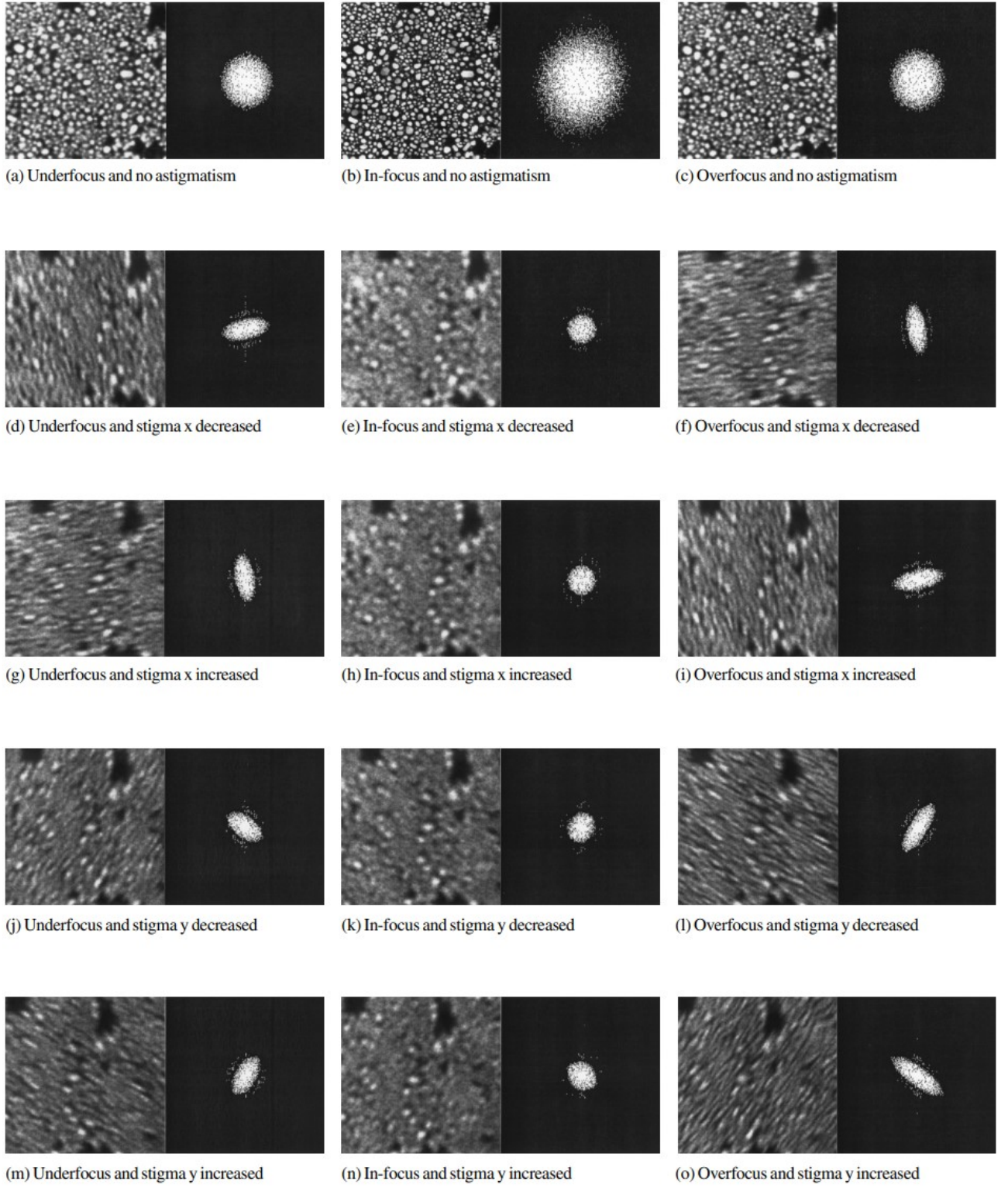
Figure 3.8: Images of gold-on-carbon sample and their fast Fourier transforms (FFTs) for different degrees of defocus and astigmatism [11].

Chapter 4

An automatic focusing and astigmatism correction algorithm for SEMs

4.1 Theory of the algorithm

K.H. Ong, J.C.H. Phang, and J.T.L. Thong proposed an algorithm for automatically correcting the focusing and astigmatism of the SEM [11], which uses the properties of the FFT of SEM images described in section 3.3.2.

Firstly, convert the FFT of the image into a binray image by applying a threshold to the magnitudes, and then segment it into eight regions as shown in Fig. 4.1. A pixel has value 1 if the magnitude is above the threshold and 0 otherwise. Let I be a matrix representing the current image with focus set to F, obtain an under-focused image I_{uf} and an over-focused image I_{of} by setting the focus to $F - \Delta F = F_{uf}$ and $F + \Delta F = F_{of}$, respectively. Let T, T_{uf} and T_{of} be matrices representing the binary FFT of I, I_{uf} and I_{of}, respectively.
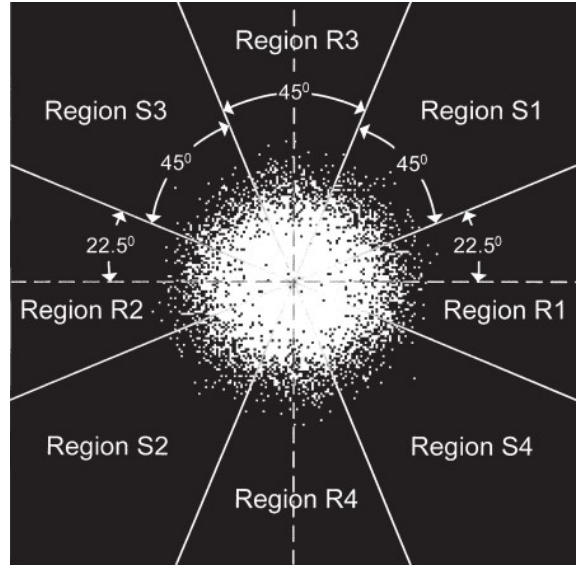

Figure 4.1: FFT segmentation [11].

The focus can be adjusted by comparing the sum of pixel values in T, T_{uf} and T_{of}. Let the perfect focus for the image be $\hat{F}$, then

$$
\begin{cases}
\text{sum}(T_{of}) < \text{sum}(T_{uf}), & \hat{F} < F_{uf} < F \\
\text{sum}(T_{of}) < \text{sum}(T_{uf}), & F_{uf} < \hat{F} < F \\
\text{sum}(T_{of}) = \text{sum}(T_{uf}), & F = \hat{F} \\
\text{sum}(T_{uf}) < \text{sum}(T_{of}), & F_{of} > \hat{F} > F \\
\text{sum}(T_{uf}) < \text{sum}(T_{of}), & \hat{F} > F_{of} > F
\end{cases}
.
$$

Let

$$
P = \text{sum}(T_{of}) - \text{sum}(T_{uf}). \qquad (4.1)
$$

Then, the rules for adjusting focus are

- when $P > 0$, the focus should be decreased.

- when $P < 0$, the focus should be increased.

After the focus has been set to the perfect value, i.e. when $F = \hat{F}$, the stigmator settings can be adjusted by comparing the sum of pixel values in different regions of T, T_{uf} and T_{of}. Let

$$
P_{R12} = \text{sum}(T_{of,R12}) - \text{sum}(T_{uf,R12}), \quad (4.2)
$$
$$
P_{R34} = \text{sum}(T_{of,R34}) - \text{sum}(T_{uf,R34}), \quad (4.3)
$$
$$
P_{S12} = \text{sum}(T_{of,S12}) - \text{sum}(T_{uf,S12}), \quad (4.4)
$$
$$
P_{S34} = \text{sum}(T_{of,S34}) - \text{sum}(T_{uf,S34}). \quad (4.5)
$$

The rules for adjusting the stigmators can be determined from Fig. 3.8. As can be seen in the figure, stigmator x affects the astigmatism in the horizontal and vertical directions, i.e. P_{R12} and P_{R34}, and stigmator y affects that in the diagonal directions, i.e. P_{S12} and P_{S34}. Then:

- when $P_{R12} > 0$ and $P_{R34} < 0$, the value of stigmator x should be decreased.

- when $P_{R12} < 0$ and $P_{R34} > 0$, the value of stigmator x should be increased.

- when $P_{S12} > 0$ and $P_{S34} < 0$, the value of stigmator y should be increased.

- when $P_{S12} < 0$ and $P_{S34} > 0$, the value of stigmator y should be decreased.

4.2 Practical considerations

The performance of the algorithm is affected by five key parameters:

- $P_{threshold}$. Ideally, the focus of the SEM should be set to a value such that $P = 0$, i.e. the under-focused image and the over-focused image have the same FFT magnitude. However, this is practically impossible due to noise, non-linearity in the lens system, etc. Therefore, $P_{threshold}$ is used to define a threshold such that the image is considered to be in perfect focus when $|P| < P_{threshold}$. If P is normalised, $P_{threshold}$ can be conveniently set to a percentage value.

- F_{step}, which is how much the focus should be adjusted during each iteration in focus adjustment. A smaller F_{step} allows the

final image to have a finer resolution but increases the amount of time needed for the adjustment. A larger F_{step} speeds up the process but may never achieve the level of focus as required by $P_{threshold}$. To achieve the best result, larger values of F_{step} should be used at the beginning and smaller ones towards the end.

- ΔF, which determines how much the focus should be changed to obtain the under-focused and over-focused images. It is more important to the astigmatism correction step. When ΔF is too small, the difference between the FFT of the under-focused and over-focused image will not be significant enough and can be easily hidden by noise. When it is too big, the FFTs will lose their high-frequency components, which again makes the difference too small.

- $S_{threshold}$. The values of P_{R12}, P_{R34}, P_{S12}, P_{S34} are prone to the influence of noise near zero. Same as the purpose of $P_{threshold}$, $S_{threshold}$ defines a threshold to circumvent this problem.

- S_{step}, which is how much the value of the stigmator control should be adjusted during each iteration in astigmatism adjustment. Similar to F_{step}, it determines the speed of the adjustment and the level of fineness of the final images.

Another factor that affects the speed of the algorithm is how fast the FFT segmentation can be done. The most straightforward way of doing it is to iterate through the whole matrix while calculating the sums in pure Python, for every FFT. This is slow, and the *Masker* class provides a faster solution.

A *Masker* is initialised with a 2D shape, and it creates eight matrices, each representing a region as shown in Fig. 4.1. For example, if the input is $[5, 7]$, the matrix created for region R1 will be

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

The code ensures that there is no overlapping between matrices apart from at the center. The sum of values in region R1 of the FFT can be calculated using:

```
ma.array(fft, masker.r1).sum()
```

4.3 Experiment, results, and discussions

Due to the impact of the COVID-19 pandemic, the algorithm has not been comprehensively tested and only some preliminary experiments were conducted to verify the logic of the algorithm and the correctness of the implementation. Fig. 4.2 shows the result of the focusing correction done by the algorithm. It was able to find the same best focus for the SEM as determined by the operator.

A major practical problem found during the experiment is that although the FFTs can be calculated within a few milliseconds using the real-time diagnostic tool, the algorithm still has to wait a few hundred milliseconds in each iteration for the SEM image to be updated. This is because the SEM needs to perform multiple scans on the object and take the average to reduce noise.

Another problem is that it is difficult to determine the best values for the five parameters mentioned earlier. The result shown in Fig. 4.2 was obtained using the best settings found empirically, which may be dependent on the model of the SEM and characteristics of the object under observation.
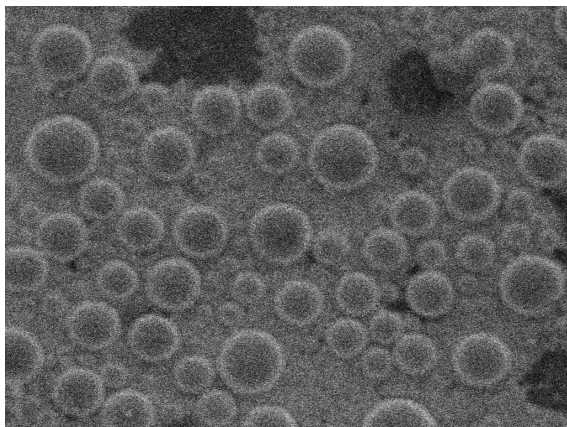
(a) Initial.
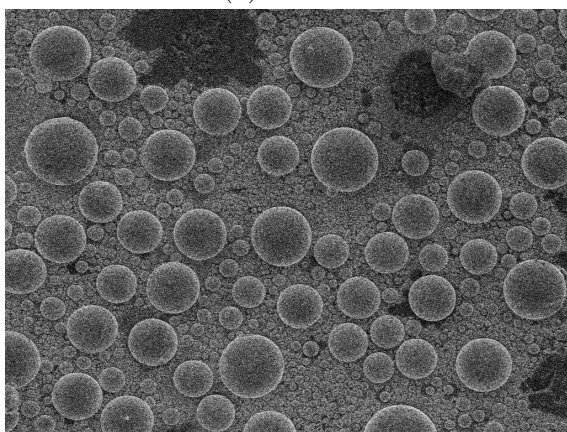
(b) Final.

Figure 4.2: Automatic focusing correction.

Chapter 5

Conclusions

The GPU can perform some general-purpose computing tasks much faster than the CPU because of its architecture that emphasises on parallelism. Experiments have shown that a common middle-range GPU can perform vector operations about eight times faster than a similarly priced CPU. This has allowed the development of a real-time diagnostic tool for SEMs.

The real-time diagnostic tool provides SEM operators a novel way for evaluating the focusing and astigmatism of SEM images, and making corresponding adjustments to the SEM. In the past, operators make judgements based on the stretching of the images. By performing the calculations ten times faster, the use of GPGPU gives operators access to the FFT of the images in real-time at about ten frames per second. This allows operators to evaluate the focusing and astigmatism of SEM images using the FFT, eliminating the subjectivity and difficulty caused by complex structures of the object in the old method.

Being real-time also means that the diagnostic tool can be used to support fast auto-matic algorithms. An algorithm has been implemented and preliminary tests have shown that it is able to correct the focusing of the SEM as good as an SEM operator. Further experiments need to be conducted to verify its ability to correct astigmatism and to further improve its speed.

Bibliography

[1] "Review SEM - What is a Review SEM?," Hitachi, [Online], available: `https://www.hitachi-hightech.com/global/products/device/semiconductor/review-sem.html.` [Accessed: 25 May 2020]

[2] "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].

[3] 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.

[4] 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.

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

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

[7] "View histograms and pixel values," Adobe, [Online], available: `https://helpx.adobe.com/uk/photoshop/using/viewing-histograms-pixel-values.html.` [Accessed: 18 May 2020].

[8] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex Fourier series," Mathematics of Computation, vol. 19, no. 90, pp. 297-297, 1965, doi: 10.1090/s0025-5718-1965-0178586-1.

[9] "2D Fourier transforms and applications," University of Oxford, 2014, [Online], available: `http://www.robots.ox.ac.uk/.` [Accessed: 19 May 2020].

[10] C. Lyman, "Correcting Astigmatism in SEM Images," Microscopy Today, vol. 27, no. 03, pp. 32-35, 2019, doi: 10.1017/s1551929519000476.

[11] K.H. Ong, J.C.H. Phang and J.T.L. Thong, "A robust focusing and astigmatism correction method for the scanning electron microscope,", scanning 19: 553-563, 1997, doi: 10.1002/sca.4950190805.