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

Liuchuyao Xu

May 24, 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 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.

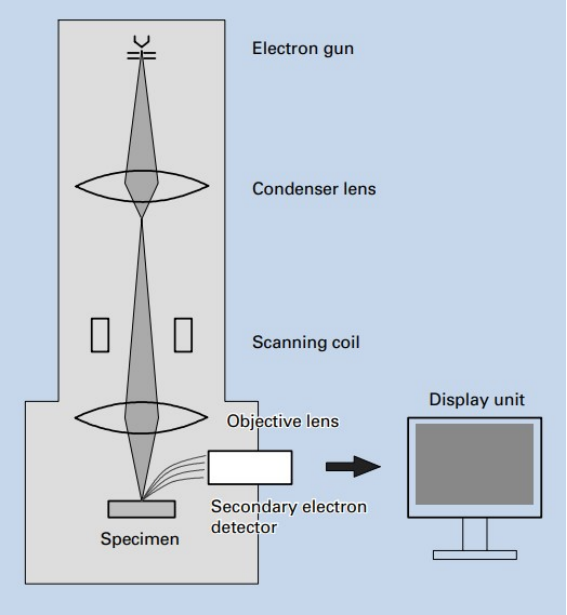


Figure 1.1: Basic construction of an SEM [3].

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

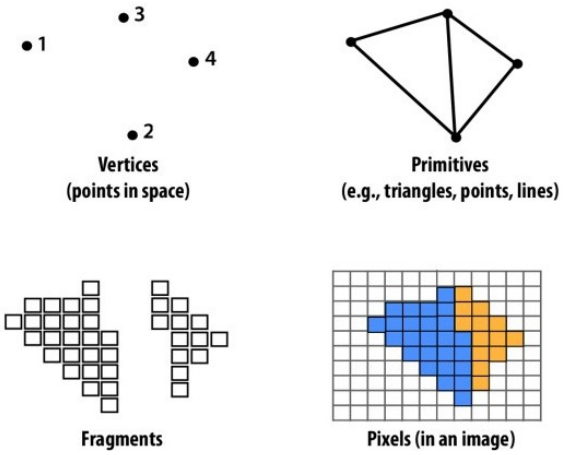


Figure 2.1: Graphics primitives [7].

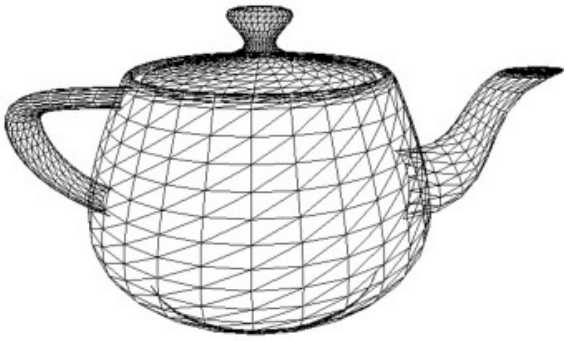


Figure 2.2: Representing an image as a 3D triangle mesh [7].

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 be-

low:

- 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.

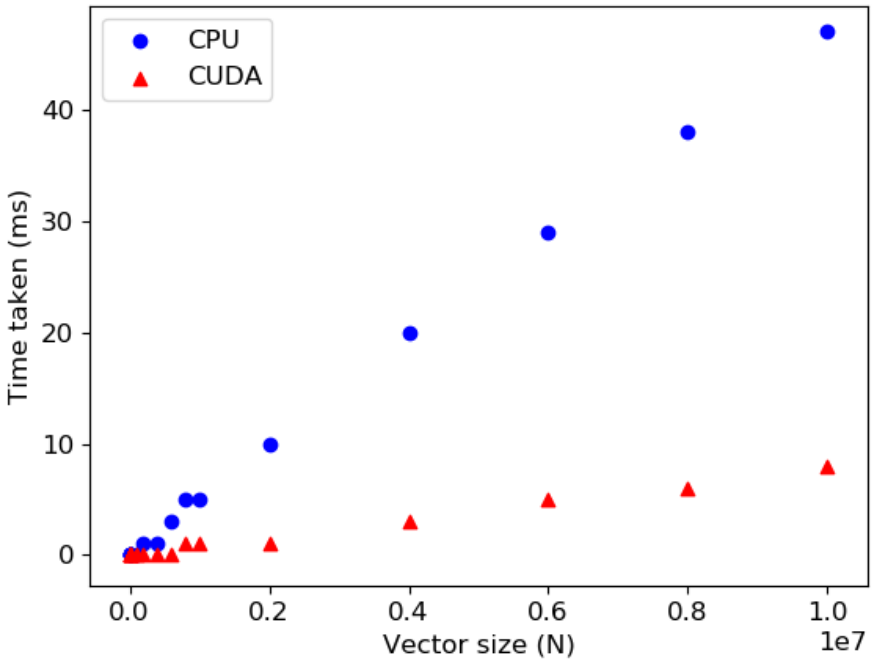
2.2 Experiment, results and discussions

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:

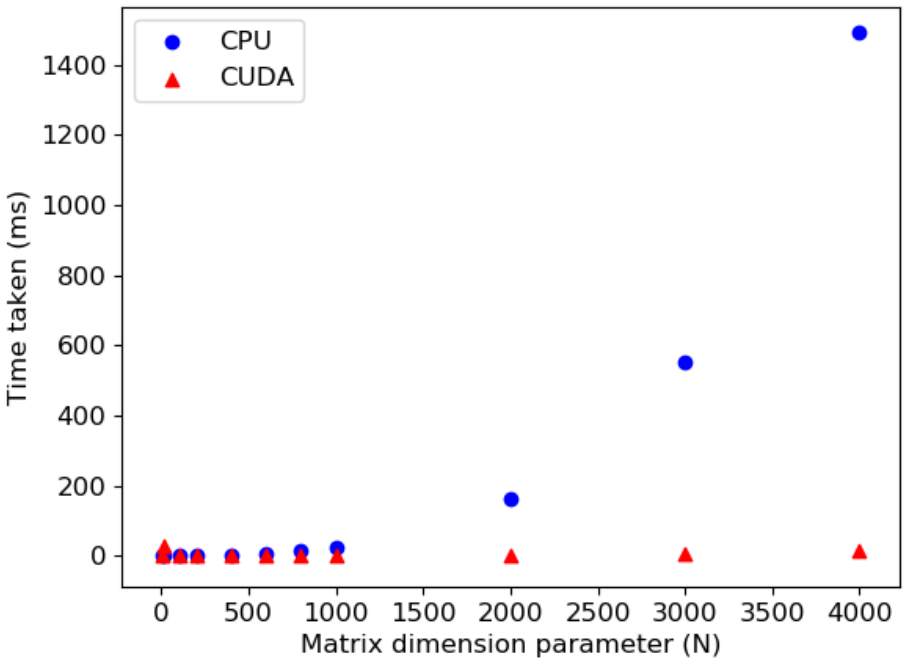
- Addition of two random vectors of size N, which has a time complexity of $O(N)$ if done without parallelism, as described in the previous section.
- Multiplication of two random matrices of dimension $N \times N$, which has a time complexity of $O(N^3)$ if done without parallelism (and without using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

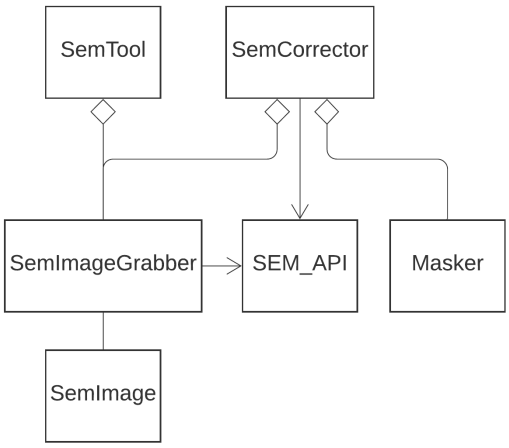


Figure 3.1: Class diagram of the software in unified modelling language (UML).

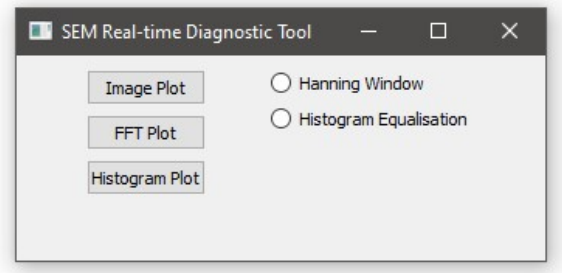


Figure 3.2: Screenshot of the GUI.

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

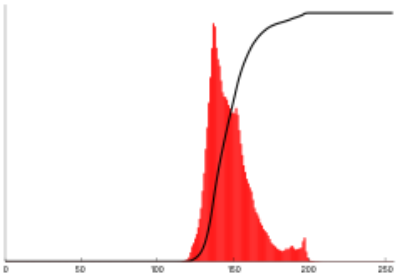
$$s_l = \sum_{i=1}^{l} n_i,$$

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

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

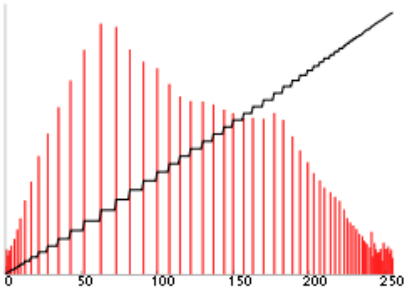
If a pixel in the original image has a grey level l, it will have a grey level f_l in the transformed image. For example, if all pixels in the original image are concentrated between grey level 100 to 200, the pixels of level 100 will become 0 in the new image while the pixels of level

(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.

that changes more abruptly will have stronger high-frequency components than a smoother one.

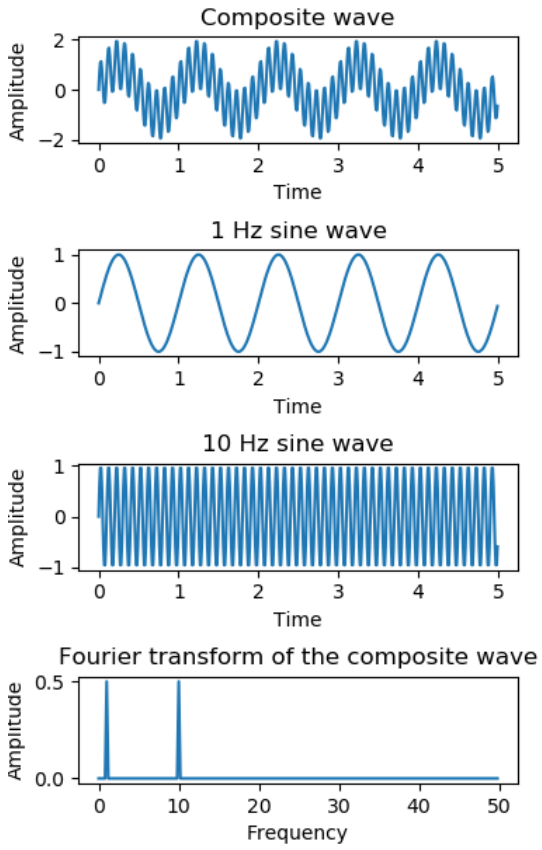


Figure 3.4: Fourier transform

3.3 Real-time fast Fourier transform calculation

3.3.1 Theory of the FFT

The Fourier transform (FT) decomposes a function into its constituent frequencies, as illustrated in Fig. 3.4. 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

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

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}, \quad 0 \leq k \leq N-1, \tag{3.3}$$

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 $0 \leq k \leq N-1$.

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

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

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. 3.5 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.

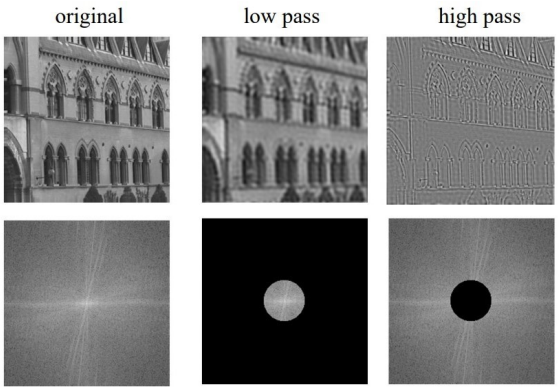
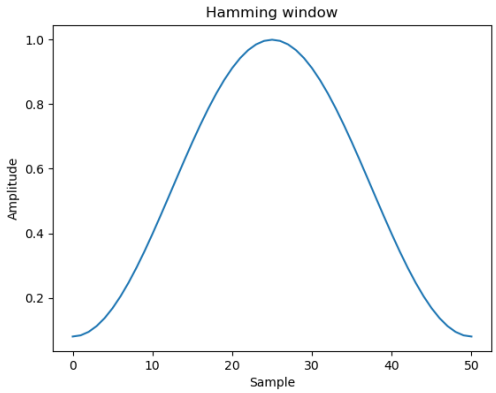


Figure 3.5: FFT and filtering on images [9].

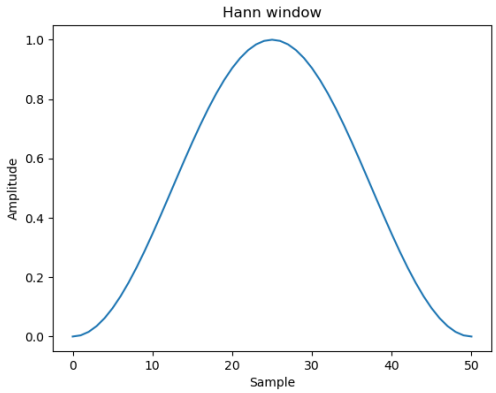
A problem in the use of the FFT for image analysis is the boundary effect. The abrupt

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. 3.6 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

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



(a) Hamming window.

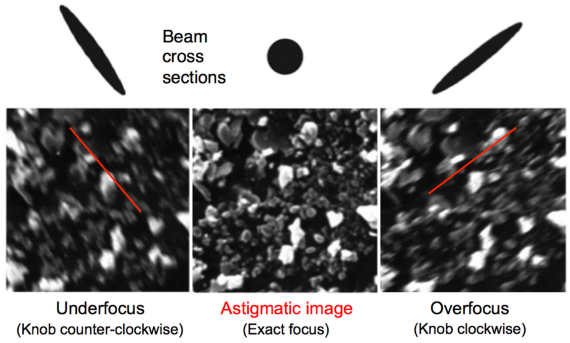


(b) Hanning window.

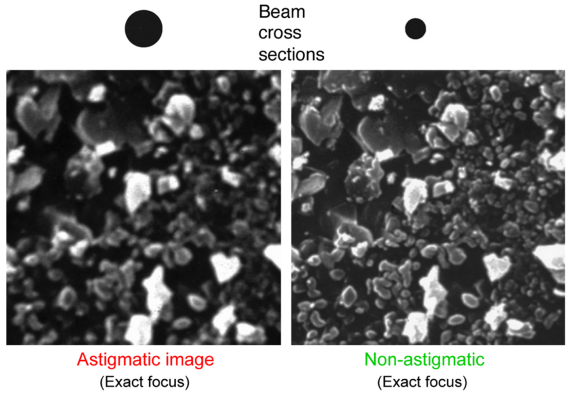
Figure 3.6: Window functions.

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. 3.7 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 3.7: 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. 3.8. 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

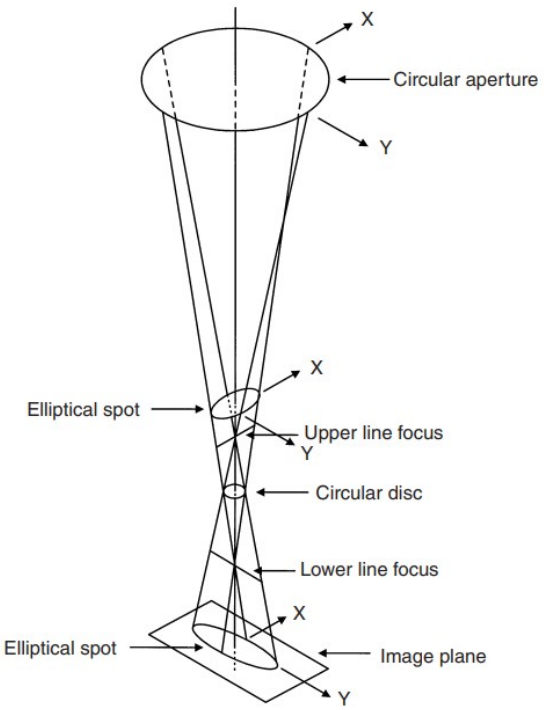
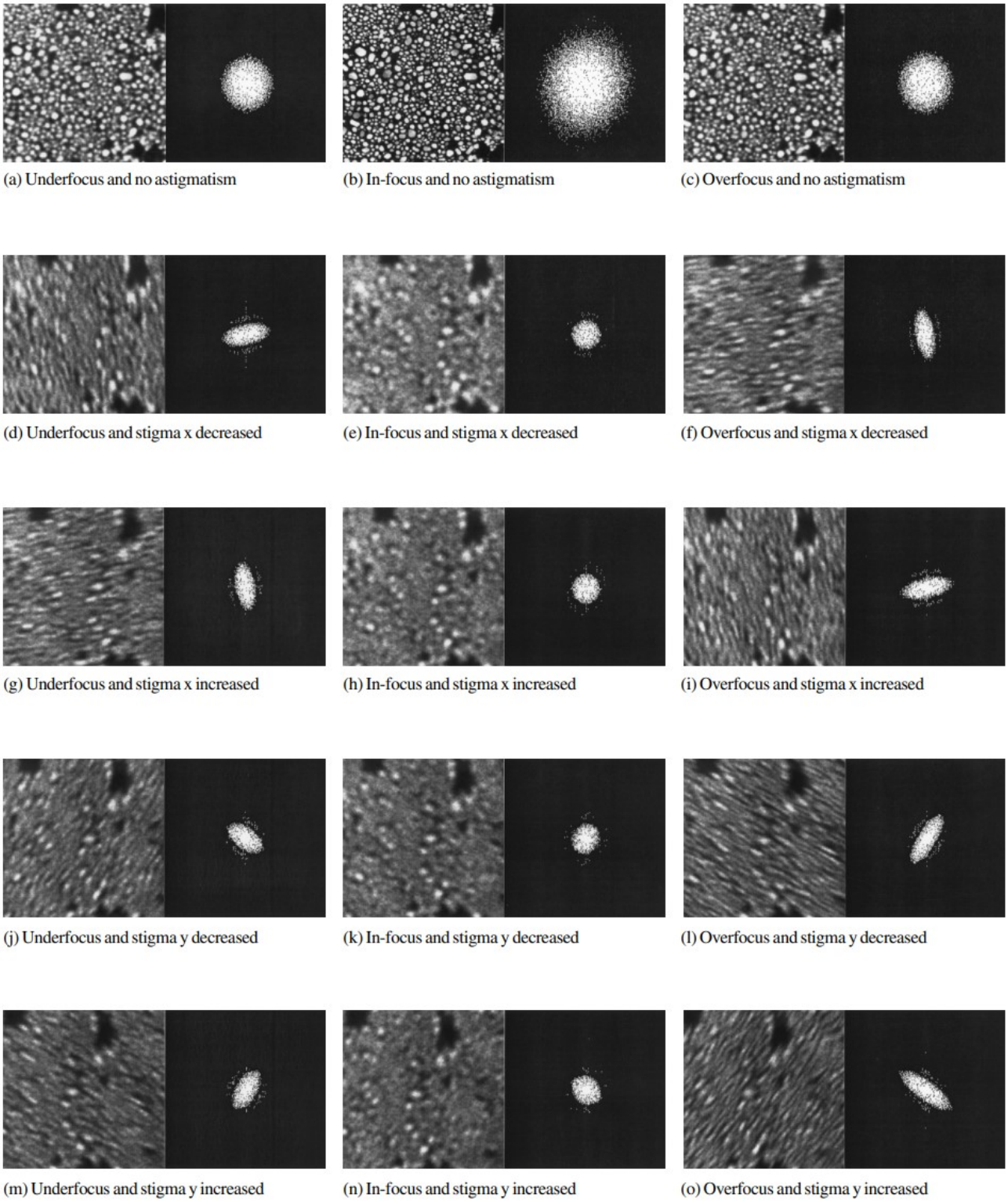


Figure 3.8: Uneven focus of the SEM.

images using the FFT. Fig. 3.9 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. 3.7).

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 3.9: Images of gold-on-carbon sample and their fast Fourier transforms (FFTs) for different degrees of defocus and astigmatism [11].

3.3.3 Performance test of the software

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.

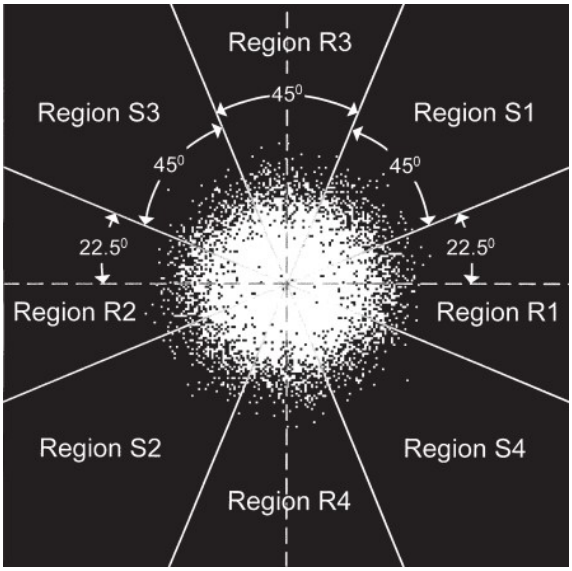


Figure 4.1: FFT segmentation [11].

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.

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

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

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

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

The rules for adjusting the stigmators can be determined from Fig. 3.9. 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

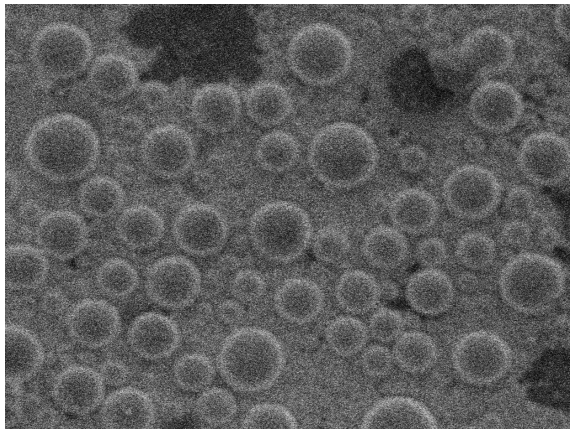
$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

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:

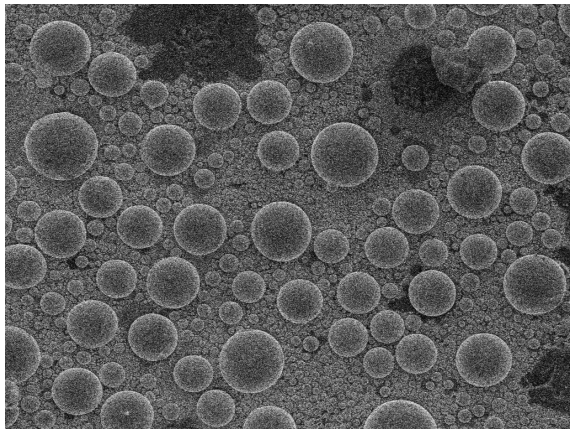
```
ma.array(fft, masker.r1).sum()
```

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) Initial.



(b) Final.

Figure 4.2: Automatic focusing correction.

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.

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 automatic 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] 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].

[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.