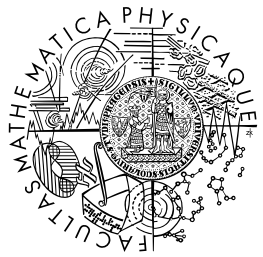


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Michal Bali

Employing GPU to Process Data from Electron Microscope

Department of Software Engineering

Supervisor of the master thesis: RNDr. Martin Kruliš, Ph.D
Study programme: Master of Computer Science
Study branch: Software Systems

Prague 2020

This is not a part of the electronic version of the thesis, do not scan!

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I thank my supervisor RNDr. Martin Kruliš, Ph.D. for constructive critic.

Special thanks go to Mgr. Jozef Veselý, Ph.D, who provided me with physics insight, reference implementation and testing data.

And also my family, Adam, Lucka and Marcel, who were there for me.

Title: Employing GPU to Process Data from Electron Microscope

Author: Michal Bali

Department: Department of Software Engineering

Supervisor: RNDr. Martin Kruliš, Ph.D, Department of Software Engineering

Abstract: Electron backscatter diffraction (EBSD) is a common tool used by physicists to examine crystalline materials, which is based on taking pictures of material microstructure using electron microscope. To determine additional characteristics of studied specimen, a specific variant called High resolution EBSD has been proposed (and partially adopted). The technique takes several subregions of the images taken by the EBSD camera and uses cross–correlation to measure deformation of obtained patterns. Usability of this method is limited by its relatively high computational complexity, which makes it useless for the analysis of larger specimen surfaces. At the same time, processing of individual subregions and images is independent, which makes it appropriate for parallelization provided by modern GPUs. In this thesis, we describe the technique used to process the EBSD data in detail, analyze it and implement the most computationally demanding parts using the CUDA technology. Compared to a reference Python implementation, we measured a speedup of 30–40-times when using a double floating precision and up to a 270-times speedup for a single precision.

Keywords: GPU parallel data image pattern

Contents

Introduction

Physicists are constantly developing new methods to describe and measure the reality around us. Nowadays, it is quite easy to collect vast amount of various data, however, it is pointless unless we are able to analyze it efficiently. This thesis aims to improve the analysis of data used to study the microstructure of materials.

Electron backscatter diffraction (EBSD) is a scientific tool used to examine crystalline materials. It is based on an electron microscope that shoots a beam of electrons towards the studied specimen, which are, in turn, reflected off the surface and captured by a camera. This results in a grayscale image of the *backscatter pattern*. Physicists then study the deformation of such patterns in order to determine several characteristics of the crystalline material, such as crystal orientation, phase or elastic strain.

In 2006, Wilkinson, Meaden and Dingley first described the technique of high resolution EBSD analysis [17]. It is based on a cross–correlation of several subregions between deformed and reference (undeformed) images. The cross-correlation is then processed so we obtain shifts between the regions of interest with subpixel accuracy by interpolating between the points of the best correlation match. From the shifts, it is possible to determine elastic strain and lattice rotation of the crystalline material.

It takes thousands or even millions of measurements to examine an object correctly using the EBSD method. Usually, the images of patterns are taken in a raster that covers the area of interest which produces thousands of them. For each image, the analysis involves cross–correlating tens of subregions (each can have roughly100×100 pixels), which is computationally expensive. At the same time, processing of individual subregions is independent, which enables data parallelism, and thus is appropriate for implementation on modern GPUs.

Faster processing of patterns enables the physicists to process bigger datasets that cover larger areas with higher resolution. Current commercially available software is insufficient for larger EBSD maps, which can contain up to a million of patterns and the analysis may take up to tens of hours. A fast GPU implementation has the potential to drastically improve the situation.

Modern EBSD cameras are also able to take hundreds of images per second, depending on the image quality. In other words, it is possible to create large datasets in a reasonable amount of time, which makes the analysis the bottleneck of the process. A reference python implementation, that has been provided by physicists from Department of Physics of Material at Charles University in Prague, can process from 5 to 50 patterns per second, depend-

ing on the input parameters.

In this thesis we describe the technique used to process the EBSD data in detail, analyze it from performance point of view, implement the most computationally expensive parts using the CUDA technology and benchmark the implementation.

The thesis is organized as follows. Chapter 1 offers a detailed explanation of the algorithm of interest. We first define the cross–correlation in section 1.2, show how it can be computed using the discrete Fourier transform. In section 1.3, we explain how to process the cross–correlation result to obtain the desired vectors of shift between the subregions. In chapter 2, we analyze the algorithm from performance point of view and outline its GPU implementation. First, we explain three GPU kernels that execute required steps (cross–correlation, sum and arg max). Then, in section 2.4 and section 2.5, we describe how they are put together to implement the algorithm. Finally, we outline how we can parallelize loading of the patterns from disk and their processing on the GPU. In chapter 3, we measure the behavior of important parts of the implementation for different parameters and compare the python and GPU implementations.

1. Processing of electron backscatter patterns

Electron microscopes are used in many scientific fields to study microstructure of materials. They rely on electrons instead of photons, which allows them to achieve greater magnification than light microscopes. The microscope emits a beam of electrons, which interact with the specimen, either transmitting through it or reflecting back. The electrons therefore carry an information about the specimen. We capture them on a phosphor screen which emits light and then we take a picture of the screen by a camera, so the result of one electron microscope measurement is an image.

One measurement of electron microscope gets information about only one microscopic point in the specimen. However sometimes we need to map a larger area of the material. For this purpose, scientists use a *scanning electron microscope*. It is a device that repeatedly emits an electron beam towards the specimen, scanning the surface in a raster pattern. So it does the measurement in one place, then moves a bit (e.g. 1 nm aside) and repeats. Since we are measuring the micro world, it takes thousands of iterations to scan a reasonably large area of the specimen. For example, test data that we used contained over 15000 images taken in a 120×120 raster covering an area of the specimen surface approximately $70\,\text{nm} \times 70\,\text{nm}$ big.

We process data obtained using the *electron backscatter diffraction* which is a scientific method that utilizes the scanning electron microscope to study crystalline materials (i.e., materials with a highly ordered microscopic structure). The principle of the electron backscatter diffraction is sketched in fig. 1.1. When a beam of electrons is emitted towards the specimen, some of the electrons backscatter, that is, they reflect back from the surface. They are then captured on a phosphor screen which is coupled with compact lens that directs the captured image towards a camera. The result of this procedure is a grayscale image of the *electron backscatter pattern*. An example of such pattern is shown in fig. 1.2a. White color in the image can be interpreted as a place where many electrons were captured whereas black areas captured smaller amount of electrons.

The electrons do not reflect randomly, but based on the examined specimen, they backscatter in a specific way, forming *Kukuchi lines* observable in the pattern. The geometry of Kukuchi lines can be interpreted as a projection of the specimen crystal structure on the flat phosphor screen.

When the crystal structure of material is changed (e.g. under stress), the deformation can be observed in the backscatter pattern as well. Therefore,

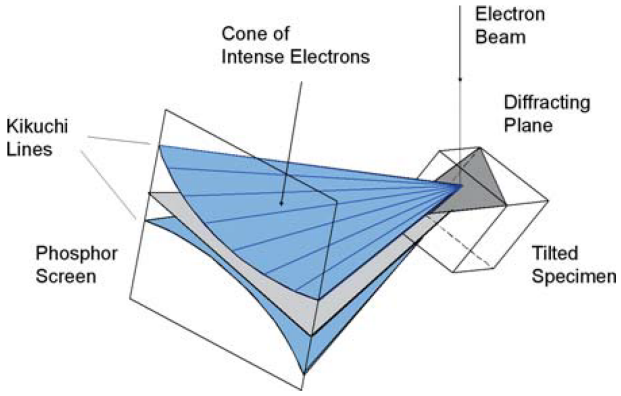
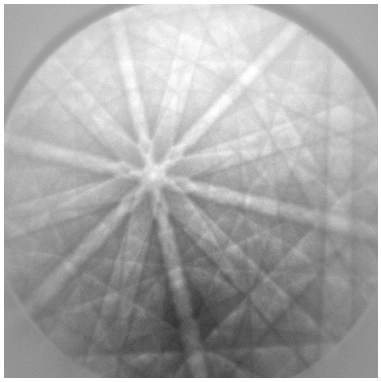
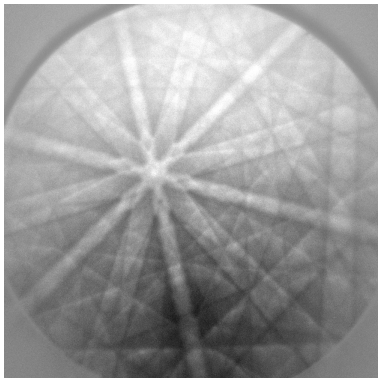


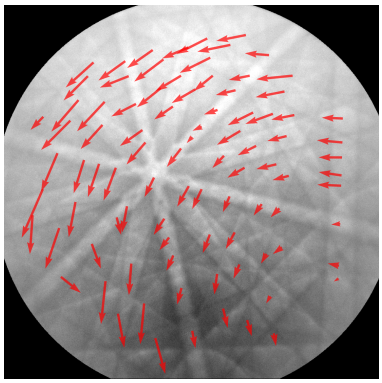
Figure 1.1: The principle of electron backscatter diffraction. [15].



(a) Initial, undeformed pattern



(b) Deformed pattern



(c) Deformation visualization

Figure 1.2: Deformed pattern with arrows visualizing the deformation. The arrows are upscaled for the visualization, because the pattern is deformed only by several pixels (the biggest offset is less than 10 pixels long, while the whole picture has resolution 873×873).

the patterns of deformed specimen can be compared to undeformed ones to measure elastic strain and crystal lattice rotation. Figure 1.2 shows a visualization of such comparison. Since different parts of the images may be deformed differently, a commonly used technique [17, 16, 2] is to choose several (usually tens to hundreds) subregions of the patterns and determine the vector of shift between the corresponding regions from both patterns. The shifts estimate the real deformation of the crystalline structure.

The comparison is done by cross–correlating respective subregions of the deformed and reference patterns. The position of the maximum value in the correlation result determines the most probable shift with one pixel precision, which is not enough. To achieve subpixel accuracy, we estimate the peak of the cross–correlation by interpolating from a small neighborhood of the maximum.

Once the shifts are computed, they are further processed to obtain an estimation of the actual crystal deformation and other characteristics. However, we do not address that part of the analysis in this thesis, as it is not nearly as computationally expensive as the processing of the images. Moreover, the comparison is commonly used in analysis of the electron backscatter diffraction data, while the processing of the shifts may be different for various purposes.

To sum up, we have thousands images of deformed electron backscatter patterns. Each of them was captured by a scanning electron microscope at a specific location of a specimen. Moreover, we have one reference image of the electron backscatter pattern. We need to quantify the deformation of each deformed pattern, so we choose several subregions and cross–correlate each respective reference and deformed subregion. Maximal correlation gives us the most probable shifts of the subregions. So for each deformed pattern, the result of our algorithm is a list of shifts corresponding to subregions. The shifts are then further processed to obtain their physical interpretation, but that is not covered in this thesis.

1.1 Algorithm description

In the rest of this chapter, we describe an algorithm used to process the electron backscatter patterns. The specifics of the algorithm correspond to a version that is used in Department of Physics of Material in Charles University in Prague.

The algorithm takes the following input and parameters:

- reference image of electron backscatter pattern with size $W_p \times H_p$

- N deformed images of electron backscatter pattern images, each with size $W_p \times H_p$
- S positions of subregions that will be compared from each image
- size $W_s \times H_s$ of each subregion
- parameter F that denotes the size of neighborhood used to interpolate the maximum with subpixel accuracy (see section 1.3)

The input images are all greyscale, which means each pixel is represented by one number. In our testing data, pixels were represented by 16 bit integers in range $0 - 65535$.

The result of the algorithm is a list of S shifts for each of the N deformed patterns (see fig. 1.2c for visualization of the shifts for one pattern).

Algorithm 1 shows a high–level pseudocode of the algorithm. It iterates through all deformed patterns and all subregion positions specified in the input. For each subregion, it computes the cross–correlation between the deformed subregion and subregion at the same position from the reference pattern. The result of cross–correlation is a matrix that expresses a measure of similarity for each possible shift between the subregions (see section 1.2). In the next step, we further process the matrix to obtain the most probable shift with subpixel accuracy (see section 1.3), which is the output of the algorithm.

Algorithm 1: Processing of electron backscatter patterns

Input: one reference electron backscatter pattern
N deformed electron backscatter patterns
position of S subregions from each pattern
size of a subregion

Output: deformation shifts for all S subregions from all N deformed patterns

```
1 refPat ← reference pattern;
2 foreach defPat in deformed patterns do
3     foreach regPos in subregion positions do
4         refReg, defReg ← extract subregions from refPat and defPat;
5         crossResult ← cross–correlate(refReg, defReg);
6         maxX, maxY ← subpixelPeak(crossResult);
7         output [maxX, maxY];
```

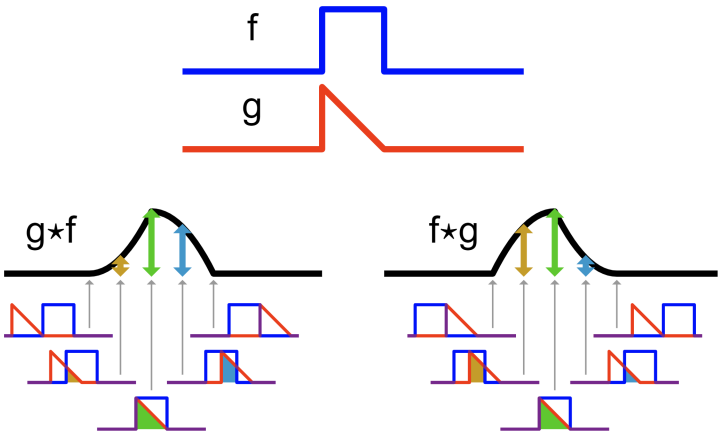


Figure 1.3: The cross–correlation of two example functions[3].

1.2 Cross–correlation

Cross–correlation provides a measure of similarity of two series for each possible shift between them. It is used in signal processing to find a shorter known feature in a signal or in image processing to locate a smaller shape in a bigger picture. In analysis of the backscatter patterns, it is used to find the most probable relative displacement between two images (subregions of images).

Cross–correlation of two discrete functions $f, g : \mathbb{Z} \to \mathbb{R}$ is a function $C : \mathbb{Z} \to \mathbb{R}$ defined as follows:

$$C[n] = (f \star g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[m+n],$$

where $\star$ denotes cross–correlation.

If $C = f \star g$ then $C[n]$ is a number that tells us how much are functions f and g similar, when g is shifted by n to the left. For each negative n, g is shifted to the right. Figure 1.3 shows two example functions and their cross–correlation.

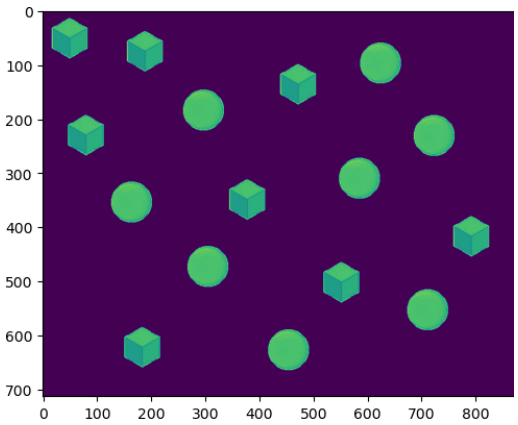
Definition of cross–correlation can be extended for two dimensions. For two–dimensional functions $f, g : \mathbb{Z}^2 \to \mathbb{R}$, their cross–correlation is a function $C : \mathbb{Z}^2 \to \mathbb{R}$ defined as:

$$C[i,j] = (f \star g)[i,j] = \sum_{x=-\infty}^{\infty} \sum_{y=-\infty}^{\infty} f[x,y]g[x+i, y+j].$$

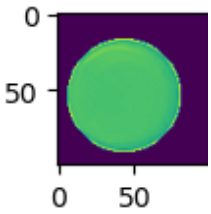
Analogously to one–dimensional cross–correlation, $(f \star g)[i,j]$ is a number that is higher if f is similar to g shifted by i horizontally and by j vertically.

Figure 1.4 demonstrates how cross–correlation can be used to search for a smaller pattern in a bigger picture. The brightest points (maxima) in fig. 1.4c correspond to the best matches between the pattern and the image.

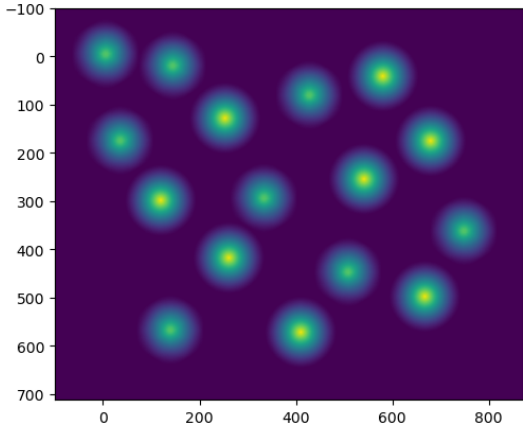
Theoretically, the domain of cross–correlation is the whole $\mathbb{Z}^2$. However, when cross–correlating two images, i.e., real–valued matrices, $(f \star g)[i, j]$ does not make sense for all $[i, j] \in \mathbb{Z}^2$, because for some of them the images are shifted so much they do not overlap. For two images with sizes $w_1 \times h_1$ and $w_2 \times h_2$, the size of their relevant cross–correlation domain is $(w_1 + w_2 - 1) \times (h_1 + h_2 - 1)$. The result is a rectangle with the following corners: $[-w_1 + 1, -h_1 + 1]$, $[-w_1 + 1, h_2 - 1]$, $[w_2 - 1, h_2 - 1]$, $[w_2 - 1, -h_1 + 1]$. Those



(a) An image



(b) Search pattern



(c) Cross–correlation of the (a) and (b) images

Figure 1.4: An example of cross–correlation used to find an object in an image. The maxima of the cross–correlation result correspond to the location of the search pattern (b) in the image (a).

are the points of cross–correlation, where the result is computed from the images overlapping only by one pixel.

1.2.1 Computing cross–correlation using discrete Fourier transform

The cross–correlation can be computed using an algorithm based on its definition with asymptotic time complexity $\mathcal{O}(W_s^2 H_s^2)$ for two images with resolution $W_s \times H_s$ — for each of the resulting $(2W_s - 1)(2H_s - 1)$ pixels, we need to multiply at most $W_s H_s$ pixels. In the following section, we describe a way to compute two–dimensional cross–correlation in $\mathcal{O}(W_s H_s \log_2(W_s H_s))$ using *discrete Fourier transform*. In the next section we show how it is used to compute *circular cross–correlation*, which can be subsequently transformed into the (non–circular) cross–correlation we defined in the previous section.

Let $N \in \mathbb{N}$, $\{x_n\} = x_0, x_1, \ldots, x_{N-1}$ and $\{y_n\} = y_0, y_1, \ldots, y_{N-1}$ be series of complex numbers. Then their circular cross–correlation is another series of N numbers defined by the formula

$$\{\mathbf{x} \star_N \mathbf{y}\}_n = \sum_{l=0}^{N-1} x_l^\star y_{(n+l) \bmod N},$$

where $\cdot \star_N \cdot$ denotes the circular cross–correlation of two series.

Circular cross–correlation can be interpreted as a cross–correlation of two periodic functions. For any periodic function with period N, we only need N consecutive values to represent, since the rest of the function repeats the same values. At the same time, given two periodic discrete functions $f, g : \mathbb{Z} \to \mathbb{C}$ with period N, their cross–correlation $f \star g$ is also periodic with the same period. So if we interpret the series in the definition of circular cross–correlation of as periodic functions, then the resulting series represents (non–circular) cross–correlation of the functions.

The circular cross–correlation can be quickly computed by using the discrete Fourier transform.

The *discrete Fourier transform* is a function $\mathcal{F} : \mathbb{C}^N \times \mathbb{C}^N$ that transforms a series of $N \in \mathbb{N}$ complex numbers $\{x_n\} = x_0, x_1, \ldots, x_{N-1}$ into another N complex numbers $\{X_n\} = X_0, X_1, \ldots, X_{N-1}$, which are defined as follows:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N} kn}.$$

The Fourier transform is invertible, so if $\mathcal{F}(\mathbf{x}) = \mathbf{X}$, then $\mathcal{F}^{-1}(\mathbf{X}) = \mathbf{x}$.

Fourier transform has a broad range of practical applications — it is used for example in digital signal processing, solving partial differential equations

or big numbers multiplication. It can be used to quickly compute the circular cross–correlation using the following theorem [14].

Let $N \in \mathbb{N}$, $\{x_n\} = x_0, x_1, \ldots, x_{N-1}$, $\{y_n\} = y_0, y_1, \ldots, y_{N-1}$ be series of complex numbers and let $\{X_n\}$, $\{Y_n\}$ be their Fourier transforms. Then, we can compute *circular cross–correlation* like so:

$$\mathcal{F}^{-1}\{\mathbf{X}^\star \cdot \mathbf{Y}\}_n = \sum_{l=0}^{N-1} x_l^\star y_{(n+l) \bmod N},$$

where $\star$ denotes the *complex conjugate* and $\cdot$ denotes element by element multiplication. Complex conjugate of a complex number $a + bi$, where $a, b \in \mathbb{R}$, is the complex number $a - bi$. Complex conjugate of series $\{a_n\}$ is series $\{b_n\}$, where every element b_n is the complex conjugate of the original element a_n.

In other words, in order to compute the circular cross–correlation of two series $\{x_n\}$ and $\{y_n\}$, we first compute their Fourier transforms $\{X_n\}$ and $\{Y_n\}$, then multiply corresponding elements of the complex conjugate of $\{X_n\}$ and $\{Y_n\}$. Finally, we do an inverse Fourier transform. We do all of this, because we are able to compute both the discrete Fourier transform and its inverse in time $\mathcal{O}(N \log N)$, which gives us the overall asymptotic time $\mathcal{O}(N \log N)$ compared to $\mathcal{O}(N^2)$ when computing the circular cross–correlation from definition.

To compute non–circular cross–correlation using this method, we use the following trick: for series $\{x_n\}$ and $\{y_n\}$ of length N, we expand both of them with N zeros and then do circular cross–correlation. The result of such operation is the desired cross–correlation, only circularly shifted by N elements. The reason why it works is illustrated in Figure 1.5. When doing regular cross–correlation (in the left side of the figure), we slide one of the series along the other and for every shift, we multiply the corresponding numbers. If a number does not have any counterpart in the other series with respect to the current shift, it is discarded. For circular cross–correlation, we again slide one of the series, but this time we do it circularly using the modulo operator. We use the zeros that we expanded the original series with to rule out the values that should not be multiplied for the current shift.

The circular cross–correlation of zero–padded series results in a series $\{c_n\} = \mathbf{x} \star_N \mathbf{y}$ of $2N$ numbers. The numbers in the series have the following interpretation: $c_0, \cdots, c_{N-1}$ are the values of non–circular cross–correlation with shifts $0, 1, \cdots, N$ and $c_{N+1}, c_{N+2}, \cdots, c_{2N-1}$ represent the shifts $(-N+1), (-N+2), \cdots, -2, -1$. c_N is always zero and does not have any interpretation.

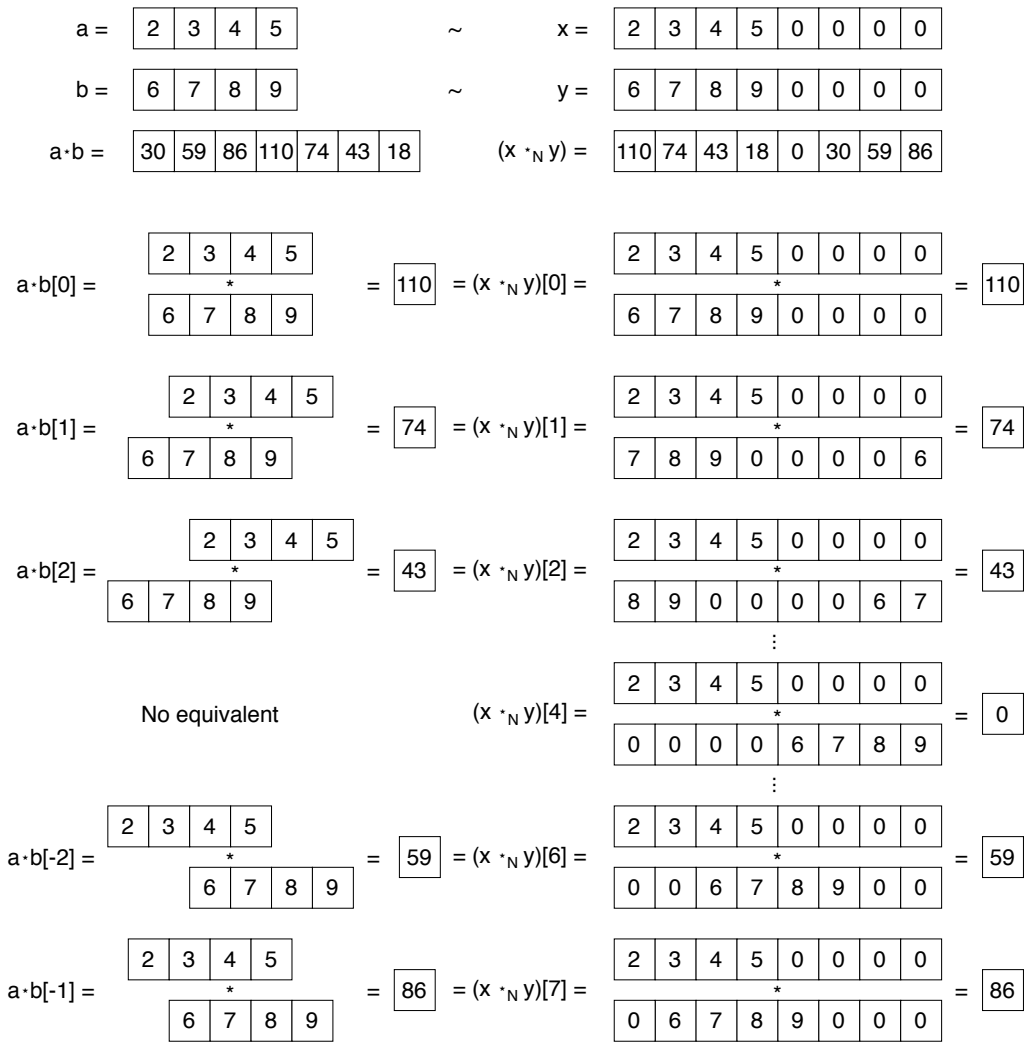


Figure 1.5: Illustration of the relationship between circular and non–circular cross–correlation. The left side shows a cross–correlation of two series a and b. The right side shows the circular cross–correlation of x and y, which are zero–padded versions of the series a and b. Corresponding values of both correlations are shown side by side.

Two–dimensional discrete Fourier transform

In order to use the described method in our implementation, we need to expand it to work on two–dimensional inputs.

Let $N, M \in \mathbb{N}$, $\mathbf{x} \in \mathbb{C}^{N \times M}$. Then $\mathbf{X} \in \mathbb{C}^{N \times M}$ is the Fourier transform of x, if the following holds for each item of the matrix $\mathbb{X}$:

$$X_{k,l} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} x_{n,m} \cdot e^{-i2\pi(\frac{kn}{N} + \frac{lm}{M})}.$$

Next, the circular cross–correlation theorem can be rewritten for two dimensions as well. Let $N, M \in \mathbb{N}$, $\mathbf{x} \in \mathbb{C}^{N \times M}$, $\mathbf{y} \in \mathbb{C}^{N \times M}$ let $\mathbf{X} = \mathcal{F}(\mathbf{x})$ and $\mathbf{Y} = \mathcal{F}(\mathbf{y})$ be their Fourier transforms. Then, we can compute the two–dimensional circular cross–correlation like so:

$$\mathcal{F}^{-1}\{\mathbf{X}^\star \cdot \mathbf{Y}\}_{n,m} = \sum_{k=0}^{N-1} \sum_{l=0}^{M-1} x^\star_{k,l} y_{(n+k) \bmod N, (m+l) \bmod M},$$

where $\cdot^\star$ denotes the complex conjugate and $\cdot$ denotes element by element multiplication.

The trick with zero–padding the inputs applies to the two–dimensional case as well. So when we want to compute the cross–correlation of two images x and y (represented by matrices in the theorem) with resolution $W \times H$, we do the following steps:

1. Zero–pad the images — we enlarge the image to the size $2W \times 2H$ by adding zeros.
2. Compute the Fourier transforms of the images, which gives us two matrices $X = \mathcal{F}(x)$ and $Y = \mathcal{F}(y)$ of complex numbers with sizes $2W \times 2H$.
3. Compute the complex conjugate of the matrix X.
4. Multiply $X^\star$ with Y element by element. The operation is also called *Hadamard product*.
5. Do the inverse discrete Fourier transform on the product.

The result is a matrix R of real numbers with the size $2W \times 2H$. It contains the cross–correlation of the input images, but it is circularly shifted. Similar to the one–dimensional case, where the item N of the series was always zero, now the column $R_{*,N}$ and row $R_{N,*}$ are both zero. They separate the matrix into 4 quadrants that are circularly shifted by N compared to the cross–correlation. Thus, the last step needed to get the resulting cross–correlation is to shift the matrix back by N, which is the same as swapping the quadrants diagonally.

1.2.2 Zero–normalized cross–correlation

Cross–correlation as we defined it (see section 1.2) has a major flaw for image processing: it is heavily affected by the brightness of the images. Consider an image, that has some subregions brighter than others. Then, by definition of cross–correlation, that subregions add more to the result, regardless of the second image. Figure 1.6a shows an example of an image that is much brighter in the top part than in the bottom. Figure 1.6c shows the result of cross–correlation with the search pattern in fig. 1.6b. It is clear that the highest values are located where the original image is brightest instead of locating the search pattern.

That is why zero–normalized cross–correlation is used. It is based on subtracting respective means from each image. Normalized cross–correlation of two images f and g is defined as follows:

$$\mathrm{ZNCC}[i,j] = \sum_{x,y} \frac{1}{\sigma_f \sigma_g} (f[x,y] - \mu_f)(g[x+i, y+j] - \mu_g),$$

where μ_f and μ_g are the means of the images and σ_f, σ_g are their standard deviations. For an image f with size $W \times H$, they are defined as follows:

$$\mu_f = \frac{1}{WH} \sum_{x,y} f[x,y],$$

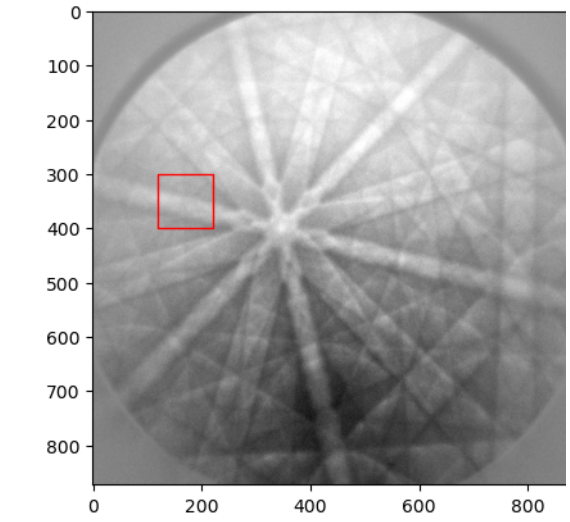
$$\sigma_f = \sqrt{\frac{1}{WH-1} \sum_{x,y} (f[x,y] - \mu_f)^2}.$$

Figure 1.6d shows the zero–normalized cross–correlation of the figures 1.6a and 1.6b. The area of the highest correlation is now clearly visible and corresponds to a Kukuchi line in the original image. Subtracting the mean of both images makes the algorithm less prone to brightness changes across the image.

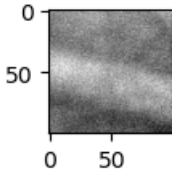
The intuition behind this behavior is as follows. By the definition of simple cross–correlation, pixels with low brightness multiply and result in low correlation. After subtracting the means, low values in both images become negative. However, two negative numbers multiply into a positive value, so they increase the correlation. On contrary, very bright and very dark pixels multiply into a negative number so their difference causes that they lower the overall correlation value.

The zero–normalized cross–correlation is used in processing of the electron backscatter patterns to compare the reference deformed subregions. It can be computed in three steps:

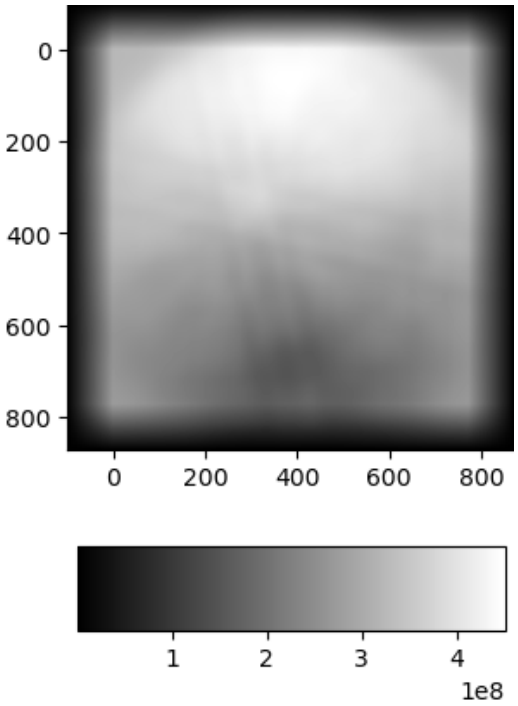
1. Find the means of the reference and deformed subregions



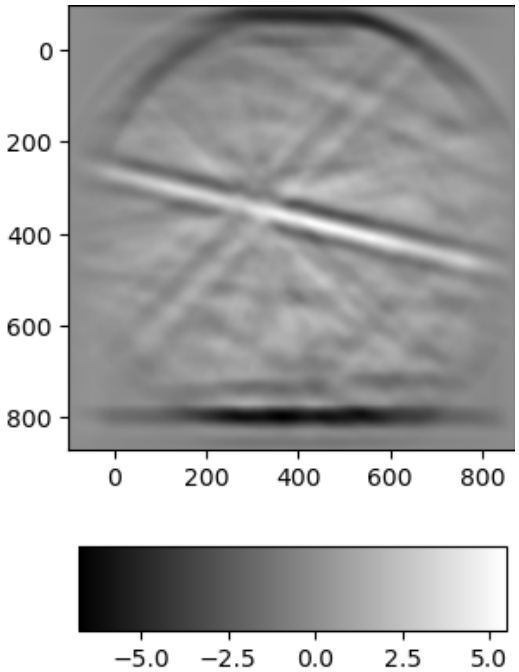
(a) An electron backscatter pattern



(b) Search pattern



(c) Simple cross–correlation



(d) Normalized cross–correlation

Figure 1.6: An image of backscatter pattern is correlated with its subregion, once using simple cross–correlation and once using the normalized cross–correlation.

2. Subtract the means from the pixels of both subregions
3. Find the standard deviations of the reference and deformed subregions
4. Divide the the pixels of both subregions by their respective standard deviations
5. Compute the cross–correlation of the subregions

First four steps normalize the subregions and then we perform the simple two–dimensional cross–correlation. The result is a two–dimensional matrix with size $(2W_s - 1) \times (2H_s - 1)$, if the subregions have size $W_s \times H_s$.

1.3 Subpixel peak

Next, we find the maximum of the cross–correlation. The location of the maximum $[x_m, y_m]$ in the matrix is the shift between the deformed and reference subregions. However, the precision of one pixel is not accurate enough in the context of the nanoworld the electron backscatter diffraction studies. That is why we use the cross–correlated data to estimate the maximum with subpixel accuracy.

We expect that if we could compute the cross–correlation with infinite resolution, so it would be continuous, the neighborhood of the maximum would look like a two–dimensional quadratic function with exactly one maximum. Since we only have limited resolution, we estimate (fit) the continuous quadratic function from several discrete points of the cross–correlation — the neighborhood of the maximum. The neighborhood is square–shaped and its size $F \times F$ is a parameter of the algorithm. Finally, we find the exact location of maximum of the estimated continuous quadratic function, which may be slightly shifted from the discrete maximum, depending on the values of the cross–correlation in the neighborhood.

So the processing of the cross–correlation is done in several steps:

1. Find the position $[x_m, y_m]$ of the maximum (often denoted as *arg max* operation) of the cross–correlation. It is a pair of whole numbers from the closed interval $(-W_s, W_s) \times (-H_s, H_s)$.
2. Use the *least squares method* to fit a quadratic function to the neighborhood of the maximum. The function is a continuous representation of the cross–correlation peak.
3. Find the position of the maximum of the continuous function.

In the following sections, we describe these steps in detail.

1.3.1 Least squares

The least squares method is used to approximate the solution of systems of equations which do not have an exact solution, since there are more equations than variables. Such systems are also called overdetermined systems. In the electron backscatter pattern analysis, the least squares method is used to approximate coefficients of a quadratic function from its several known points.

In the first place, let us formulate the problem that the least squares method addresses. Let $n, m \in \mathbb{N}, n > m; A \in \mathbb{R}^{n \times m}; x \in \mathbb{R}^m$ and $b \in \mathbb{R}^n$. Then $Ax = b$ is an overdetermined system of linear equations. It may have a solution, if b is a linear combination of column vectors of A (b is in column space of A), but in general, that is not the case.

Instead of solving the system, we search for a vector x' that has the least difference from the right hand side of the equation:

$$x' = \min_x \|Ax - b\|,$$

where $\|\cdot\|$ denotes Euclidean norm. For a vector $x \in \mathbb{R}^n$, it is defined as:

$$\|x\| = \sqrt{\sum_{i=0}^{n} x_i^2}.$$

So we minimize the sum of squared differences between the left and right sides of the equations in the system, hence the name least squares.

The least squares solution can be calculated using only matrix multiplication and inversion as follows [1]:

$$x' = (A^T A)^{-1} A^T b.$$

This formula is not used in practice because of effectiveness and numerical stability (instead, it is better to solve the equation $A^T Ax = A^T b$). However, for the algorithm described in this thesis, the approach is sufficient, since the matrix A is constant in our case (as we will explain in the end of section 1.3.2 for further explanation).

1.3.2 Maximum neighborhood fitting

We use the least squares method to estimate the coefficients of a two–dimensional quadratic function from the neighborhood of the cross–correlation maximum. The neighborhood has size $F \times F$ and has the shape of a

square, so for maximum at location $[x_m, y_m]$, the neighborhood points are at positions

$$\{x_m - \frac{F-1}{2}, \cdots, x_m, \cdots, x_m + \frac{F-1}{2}\} \times \{y_m - \frac{F-1}{2}, \cdots, y_m, \cdots, y_m + \frac{F-1}{2}\}.$$

We only consider odd F, so the maximum is always in the middle.

The estimated two–dimensional quadratic function $f : \mathbb{R} \to \mathbb{R}$ can be written as as:

$$f(x, y) = q_1 + q_2 x + q_3 y + q_4 x^2 + q_5 xy + q_6 y^2,$$

where $q_1 \dots q_6 \in \mathbb{R}$. To estimate such function is to find the 6 coefficients $q_1 \dots q_6$. From the known points of the neighborhood N of size $F \times F$, it is possible to create the following system of equations:

$$\forall [x, y] \in \{0, 1, \dots, F-1\}^2 : q_1 + x q_2 + y q_3 + x^2 q_4 + xy q_5 + y^2 q_6 = N[x, y],$$

where $q_1 \dots q_6$ are unknown variables.

So for each point with coordinates $[x, y]$ in the neighborhood, we create one equation with substituted x and y. We have F^2 known points, from which we create s^2 equations. We have 6 unknown variables and F has to be greater than one and odd, so we have more equations than unknowns.

Example of creating equation system

For instance, let F be 3 and N be the neighborhood of the maximum:

$$N = \begin{bmatrix} 751 & 819 & 765 \\ 825 & 934 & 810 \\ 768 & 798 & 725 \end{bmatrix}.$$

Then, the linear system looks like this:

$$\begin{aligned}
x = 0, y = 0 &\to q_1 + 0q_2 + 0q_3 + 0q_4 + 0q_5 + 0q_6 = N[0, 0] = 751 \\
x = 1, y = 0 &\to q_1 + 1q_2 + 0q_3 + 1q_4 + 0q_5 + 0q_6 = N[1, 0] = 819 \\
x = 2, y = 0 &\to q_1 + 2q_2 + 0q_3 + 4q_4 + 0q_5 + 0q_6 = N[2, 0] = 765 \\
x = 0, y = 1 &\to q_1 + 0q_2 + 1q_3 + 0q_4 + 0q_5 + 1q_6 = N[0, 1] = 825 \\
x = 1, y = 1 &\to q_1 + 1q_2 + 1q_3 + 1q_4 + 1q_5 + 1q_6 = N[1, 1] = 934 \\
x = 2, y = 1 &\to q_1 + 2q_2 + 0q_3 + 4q_4 + 2q_5 + 1q_6 = N[2, 1] = 810 \\
x = 0, y = 2 &\to q_1 + 0q_2 + 2q_3 + 0q_4 + 0q_5 + 4q_6 = N[0, 2] = 768 \\
x = 1, y = 2 &\to q_1 + 1q_2 + 2q_3 + 1q_4 + 2q_5 + 4q_6 = N[1, 2] = 798 \\
x = 2, y = 2 &\to q_1 + 2q_2 + 2q_3 + 4q_4 + 4q_5 + 4q_6 = N[2, 2] = 725
\end{aligned}$$

The left sides contains the equation of the two–dimensional quadratic function with substituted points from the set $\{0, 1, 2\}^2$ (for this example, we have chosen the zero point to be the top left corner). The right side contains all the corresponding values from neighborhood N.

Estimation of function coefficients using least squares method

Once we create the system of equations, we solve it using the least squares method. Recall its solution:

$$x' = (A^T A)^{-1} A^T b.$$

Matrix A in the solution corresponds to the left sides of the system of equations, b is the vector with values of the neighborhood. We can observe that the left sides of the equations (matrix A) are constant for any neighborhood of size F. That also implies that the expression $(A^T A)^{-1} A^T$ in the least squares solution is constant for the values of the neighborhood and only depends on the parameter F.

That means we can precompute the constant part of the least squares formula for each F. That reduces the computation of least squares to only one matrix–vector multiplication. We do not expect that F can have many different values. It has to be odd and the neighborhood is expected to be fairly small — for this thesis we implemented only the options $F \in \{3, 5, 7, 9\}$.

1.3.3 Maximum of fitted function

The last step in the algorithm is to find the maximum of the quadratic function. We do that using standard methods of mathematical analysis: we find where are the partial derivations of the function equal to zero.

The partial derivations of the quadratic function are:

$$\frac{\partial f}{\partial x}(x, y) = 2q_4 x + q_5 y + q_2,$$
$$\frac{\partial f}{\partial y}(x, y) = q_5 x + 2q_6 y + q_3.$$

If the function has an extreme, then it is located at the point where the partial derivations are both zero. That implies a linear system of equations with the following solution:

$$x_m = \frac{2q_2 q_6 - q_3 q_5}{q_5^2 - 4q_4 q_6}$$
$$y_m = \frac{2q_3 q_4 - q_2 q_5}{q_5^2 - 4q_4 q_6}$$

The point $[x_m, y_m]$ is the maximum of the function fitted to the neighborhood of the cross–correlation maximum.

Scaling the neighborhood by constant

When we multiply the result of the cross–correlation by constant, the result of the algorithm does not change. The reason is as follows. In the first step of the algorithm, we find the position of maximum of the cross–correlation which is not affected by constant scaling. Next, since the neighborhood is scaled by constant, the quadratic function approximation will be scaled as well. Finally, we find the maximum position of the estimated continuous function, which is the same as if the function was not scaled.

It means that when computing the zero–normalized cross–correlation, we can omit two of its steps — computation of standard deviation and division by the result, which simplifies the final algorithm.

1.4 Summary

Algorithm 2 summarizes the algorithm described in this chapter. It is clear that all the patterns and even their subregions are processed separately so it is easy to parallelize it. At the same time, there may be thousands of the patterns and tens of subregions in each, so there is enough data to utilize the whole GPU. The most computationally expensive part is the cross–correlation. We explain the implementation details of the algorithm in the next chapter.

Algorithm 2: EBSP deformation processing

Input: one reference electron backscatter pattern
N deformed electron backscatter patterns
position of M subregions from each pattern
size of a subregion
parameter F — size of neighborhood

Output: deformation shifts for all M subregions from all N deformed patterns

```
1  refPat ← load reference pattern from disk;
2  foreach deformed pattern do
3      defEBSP ← load deformed pattern from disk;
4      foreach regPos in subregion positions do
5          refReg ← extract subregion at regPos from refPat;
6          defReg ← extract subregion at regPos from defPat;
7          refMean, defMean ← means of both subregions;
8          subtract refMean from pixels of refReg;
9          subtract defMean from pixels of defReg;
10         crossResult ← cross–correlate(refReg,defReg);
11         maxX, maxY ← subpixelPeak(crossResult);
12         output [maxX, maxY];
13 Function subpixelPeak(crossResult)
14     [maxX, maxY] ← argmax(crossResult);
15     neighbors ← extract neighborhood of [maxX, maxY];
16     coefs ← fitQuadraticFunction(F, neighbors);
17     return arg max of the quadratic function with coefficients coefs ;
```

2. Analysis and Implementation

In this chapter we analyze the algorithm used to process EBSD data and explain how to implement it effectively for GPUs. We use the CUDA platform, which is currently the most popular technology for general purpose computing on graphics cards.

The input for the algorithm consists of one reference and many deformed backscatter patterns which are captured in greyscale images. There may be up to tens of thousands of them. In general, the format of the pictures is not important, as long as it is possible to load them from disk quickly. For example, our testing data consists of 15000 images saved in TIFF format without any compression. Each picture has resolution of approximately 900×900 pixels and each pixel is represented by a 16 bit unsigned integer — the higher the integer, the higher is the luminosity of the pixel.

The result of the algorithm is a list of two–dimensional vectors that we write to the standard output.

There are further parameters to the algorithm explained in the chapter 1. Table 2.1 summarizes them along with the range of values that we consider. The size of input pattern range is based on the resolution of commonly accessible EBSD cameras [5] [6]. Our testing data resolution is also within this range. The range of rest of the parameters is based on consultation with physicists from Department of Physics of Material in Charles University in Prague.

In the following sections, we first explain the computation of cross–correlation, sum and arg max on the GPU. Then, algorithm, we explain how they are used together to implement the algorithm outlined in chapter 1. We can divide it into 2 major parts: first, preprocessing of the images prior to cross–correlation that includes loading the image from disk and subtracting sums

Parameter	Label	Considered value range
Size of input pattern	$W_p \times H_p$	640×480 – 1400×1000
Number of subregions	S	20 – 120
Size of a subregion	$W_r \times H_r$	40×40 – 250×250
Diameter of a neighborhood	F	3,5,7,9

Table 2.1: Summary of algorithm parameters with example values that show their order of magnitude.

of the subregions, which is addressed in section 2.4. Second part, described in section 2.5, consists of processing of its result, which includes search of the maximum position and finding the maximum within each neighborhood.

The most important feature of the algorithm upon which we build our GPU implementation is the fact that all the subregions are completely independent from each other. So every kernel described in the following sections processes all the subregions of a pattern in parallel (we also implemented batch parameter described in section 2.7, which allows processing more images at once). This way, we can make sure we utilize the whole GPU.

2.1 Cross–correlation

We start by analyzing the core of the algorithm — cross–correlation, which is also the most computationally demanding part. The input for cross–correlation are two images, in our case it is a deformed and a reference subregion. More precisely, there are several pairs of subregions, but since they are completely independent, we explain the algorithm for one pair only. With respect to the definition in section 1.2, we cross–correlate the reference with deformed subregion, i.e. $reference \star deformed$, not the other way around. Also, both subregions are of the same size, which simplifies some aspects of the algorithm.

2.1.1 Related work

Since cross–correlation is quite a common operation in image processing, various researchers implemented it on GPUs. For example, Liu, Zou and Luo implemented the cross–correlation for CUDA 3.1 using the cuFFT library in 2011 [11]. Lewis explained how to optimize computation of normalized cross–correlation denominator [9] and Gangodkar et al. implemented it for GPU [7]. It is also used as part of various image processing algorithms, for example Idzenga, Gaburov, Vermin, Menssen and De Korte used a GPU implementation of normalized cross–correlation for ultrasound images analysis [8].

There are also libraries that can leverage GPUs to compute cross–correlation. Matlab has a function called `xcorr2` that can be GPU–accelerated [13]. The NPP library with C interface by NVIDIA [4] also contains a function to compute two–dimensional cross–correlation. However, none of the libraries we managed to find offers a possibility to process batches of images. Since we process a large amount of rather small images, there would be unnecessary overhead of many library function calls that do only a small amount of work.

2.1.2 Definition–based algorithm

We first describe a naive algorithm designed directly from the definition. Then we explain another one that uses discrete Fourier transform. The reason why we implement two versions is that although the Fourier transform based algorithm has better asymptotic complexity, the subregions may not be big enough to reflect it.

A serial algorithm for cross–correlation is shown in Algorithm 3. It is directly based on the definition. It iterates over all possible shifts between the images (subregions), i.e. $\forall[shiftX, shiftY] \in [-W_s+1, W_s-1] \times (-H_s, H_s)$. For each of the shifts $[shiftX, shiftY]$, we sum over the products of pixels that overlap when we shift the deformed subregion by shiftX pixels horizontally and by shiftY vertically.

Algorithm 3: Serial algorithm that computes cross–correlation.

Input: reference: an array of pixels of a reference subregion
deformed: an array of pixels of a deformed subregion
W_s, H_s: size of both subregions
Output: result: cross–correlation between reference and deformed subregions

```
1  for shiftX ∈ [−W_s + 1, W_s − 1] do
2      for shiftY ∈ [−H_s + 1, H_s − 1] do
3          sum = 0;
4          for x ∈ [0, W_s − 1] do
5              for y ∈ [0, H_s − 1] do
6                  shiftedX = x + shiftX;
7                  shiftedY = y + shiftY;
8                  if shiftedX ∈ [0, W_s − 1] and shiftedY ∈ [0, H_s − 1]
                    then
9                      sum += reference[x,y] *
                        deformed[shiftedX, shiftedY];
10         result[shiftX, shiftY] = sum;
```

The inner two loops of the algorithm iterate through all pixels of the reference image. However that is not necessary for all shifts, since for most of them only smaller parts of the images overlap (the only shift that requires iteration through all points is $[0, 0]$). The if statement then filters out the pixels that do not overlap for specific shift. For performance reasons, we can

Algorithm 4: Pseudocode of CUDA kernel that computes cross–correlation.

```
1  shiftX = threadId.x;
2  shiftY = threadId.y;
3  intervalX = [max(−shiftX, 0), min(W_s − shiftX, W_s) − 1];
4  intervalY = [max(−shiftY, 0), min(H_s − shiftY, H_s) − 1];
5  sum = 0;
6  for x ∈ [max(−shiftX, 0), min(W_s − shiftX, W_s) − 1] do
7      for y ∈ [max(−shiftY, 0), min(H_s − shiftY, H_s) − 1]+ do
8          shiftedX = x + shiftX;
9          shiftedY = y + shiftY;
10         sum += reference[x,y] * deformed[shiftedX, shiftedY];
11 result[shiftX, shiftY] = sum;
```

get rid of the if statement, if we rewrite the two inner loops to always stay within the boundaries of the images.

We will only reason about $shiftX$, since the same argumentation applies to $shiftY$. So we need to find out for which $x \in [0, W_s - 1]$ holds that $(x + shiftX) \in [0, W_s - 1]$, if $shiftX \in [-W_s + 1, W_s - 1]$. The result is the following interval:

$$x \in [max(-shiftX, 0), min(W_s - shiftX, W_s) - 1].$$

This gives us an interval through which we iterate x in the inner loops for each *shiftX*. Similar modification applies for the y loop based on $shiftY$ as well.

The modified algorithm written as a CUDA kernel is listed in algorithm 4. We implemented the algorithm for CUDA by parallelizing over the outer two loops. That means each thread computes one shift and thus one value of the result. It uses two–dimensional blocks, so we can just use the two indices as *shiftX* and *shiftY*. For computing more subregions at a time, we simply use more threads.

2.1.3 The cuFFT library

We now move on to explain our implementation of the cross–correlation computation using the Fourier transform. We use the cuFFT[1] library for CUDA to compute the discrete Fourier transform and its inverse. It is a highly optimized implementation of the *Fast Fourier algorithm*, which computes the

[1] `https://docs.nvidia.com/cuda/cufft/index.html`

Fourier transform in time $\mathcal{O}(n \log n)$. It also supports batched transformations (i.e. several unrelated transformations can be done by calling a single library function), which is very important for our implementation, since we do many transformations of rather small images at once. In this section, we explain the implementation for one subregion only, since it is trivial to expand it for more subregions.

There are two functions in the cuFFT library that are essential for us `R2C` and `C2R`, which compute the discrete Fourier transform and its inverse, respectively. `R2C` takes an array of real numbers and computes their discrete Fourier transform, outputting an array of complex numbers. A complex number is represented as a pair of floats/doubles. The `C2R` function takes an array of complex numbers, computes their inverse Fourier transform and outputs an array of real numbers.

Both of the functions use an important property of Fourier transform: A series x of $N \in \mathbb{N}$ numbers is real-valued if and only if the Fourier transform of x denoted as X satisfies the Hermitian symmetry, e.g. $X_k = X^{\star}_{N-k}$. So it is enough to store just half of the transformed array, since the second half can be trivially computed. cuFFT does just that. Analogous theorem applies to two–dimensional Fourier transform as well, so cuFFT operates on roughly half of the elements, namely on the elements with index from the following set: $\{0, 1, \cdots, \lfloor W/2 \rfloor + 1\} \times \{0, 1, \cdots, H\}$, where H is number of rows and W is number of columns of the input matrix.

It also makes it possible to store the Fourier transform in roughly the same array as the input and thus perform in–place transforms. The output consists of complex numbers, which are represented by two real numbers, i.e., one element takes twice as much bytes. But at the same time, we only need $W/2 + 1$ columns (we can remove the floor operator, because W is always even in our context as we double the size of a subregion with zero padding). The result is that we save the input in a real matrix with size $(W + 2) \times H$, with the two extra columns unused. That is the same amount of space that we need for the transformed complex matrix with size $W/2 + 1 \times H$.

2.1.4 Using cuFFT to compute cross–correlation

We need to cross–correlate a reference and a deformed image. Recall that it can be computed in several steps explained in section 1.2.1:

1. Zero–pad the images.

2. Compute the Fourier transforms of the images, which gives us two matrices $X = \mathcal{F}(x)$ and $Y = \mathcal{F}(y)$.

3. Multiply $X^\star$ with Y element by element.

4. Do the inverse discrete Fourier transform on the product.

5. Swap quadrants of the result.

We use cuFFT functions described in previous section to perform steps 2 and 4. The step 3 requires a custom kernel. The first and the last step depend on the remaining ones and we show how to optimize them out below.

We have only one reference image that is cross–correlated with all the deformed ones. So we can prepare the reference subregions by executing the first two steps only once and just read from the prepared buffer in step 3 during processing of all the deformed images.

The cuFFT functions have both an in–place version, which is used when the input and output buffers are the same, and out–of–place version, which is used for different input and output buffers. Using the out–of–place has the disadvantage if allocating multiple buffers — one for input zero–padded data, one for transformed matrix and one for the output. However there are two advantages: first, the out–of–place versions of the cuFFT functions proved to be slightly faster. Second, it is possible to preserve the zero padding created by the first step when using the out–of–place version. The zero–padding can be written only once for the first image and then every time we load a new image (prior to the cross–correlation computation), we overwrite the same portion of the buffer with new deformed subregion. All the subregions are of the same size, so the same portion of the buffer is used every time. Since memory consumption is not an issue with our implementation, we can afford to allocate more buffers so we consider the out–of–place version better.

For the step 3, we implemented a kernel that does the element–wise multiplication between the complex conjugate of the Fourier–transformed reference subregion and the Fourier transform of deformed subregion. The result of the multiplication can be outputted to the buffer with Fourier transform of the deformed subregions, since we do not need it any more. Implementation of the kernel is quite straightforward, each thread loads two complex numbers, multiplies them and writes them back. It can benefit from the batch parameter (see section 2.7) — since we process several deformed subregions in one kernel call, we can load the complex number from reference subregion only once and use it multiple times.

Then we compute the inverse Fourier transformation using the `C2R` function from the cuFFT library. The result is the cross–correlation, but with extra row and column of zeros and swapped quadrants as described in section 1.2.1.

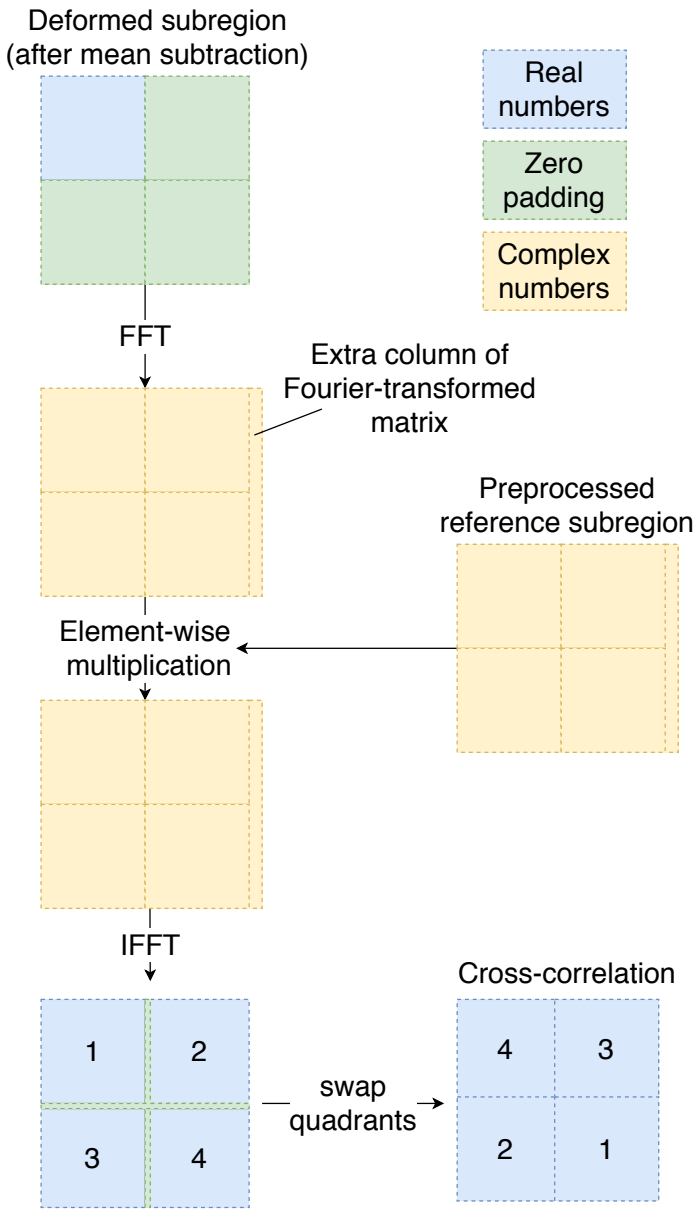


Figure 2.1: The process of computing cross–correlation using the discrete Fourier transform.

Figure 2.1 shows all the steps with illustration of used buffers. The input image has size $W_s \times H_s$, but it has to be inside a buffer with size $2W_s \times 2H_s$ to accommodate the zero padding. The transformed image has the size $(W_s + 1) \times 2H_s$ complex numbers, which is the same as $(2W_s + 2) \times 2H_s$ of real numbers. The same–sized buffer is needed after the multiplication. Then we perform the inverse Fourier transform to get the cross–correlation with swapped quadrants (as described in section 1.2.1).

The last step to get the cross–correlation is to swap the quadrants of the result. The most straightforward solution is to start a kernel which does just that and thus finalizes the cross–correlation. Such kernel moves a considerable amount of data around and thus is relatively expensive. Instead, we can modify the rest of the implementation, so it accesses the data in corresponding way. In the next steps, we need to find the position of the maximal value in the cross–correlation and then process the neighborhood of the maximum. It can be easily modified to work with incomplete cross–correlation with swapped quadrants. The search of maximum stays the same and outputs the position of maximum, which then has to be interpreted correctly when fetching the neighborhoods to process them. This trick removes the overhead of running the extra kernel at the cost of more complicated, but only slightly more demanding data address calculation. Moreover, it works with much less amount of data, as the neighborhoods are only a small part of the cross–correlation.

To sum up, we have described two ways to compute cross–correlation. The first one is based on the definition, the second one uses the Fourier transform which makes it possible to achieve better asymptotic time complexity. However for smaller input sizes, the definition–based approach is faster — we measure that in chapter 3.

2.2 Sum computation

Computation of sum is a textbook example of parallel reduction. In [12], Justin Luitjens explains how to implement it on modern GPUs. Since it is very expensive to communicate between arbitrary threads during computation, the reduction is separated into three steps: first, each thread loads several values and computes their sum, then we reduce the data within each thread block individually and finally we synchronize single value for each block. The major difference for our case is that we compute a sum for each of several subregions, instead of reducing over the whole input data. The problem is that we cannot load values of two different subregions into one thread block. We solve this by assigning whole blocks to the subregions

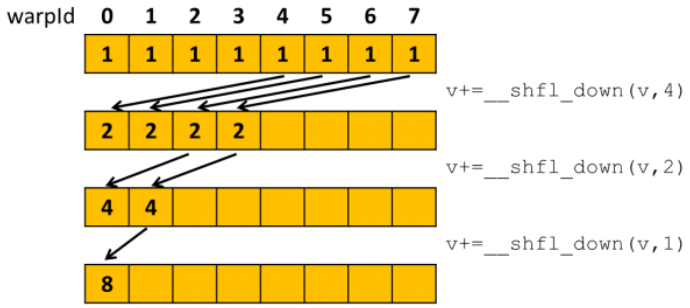


Figure 2.2: An example of warp–level reduction for 8 threads [12]. *warpId* is a number of thread within its warp.

rather than just assigning threads to pixels.

Let K be number of values that each thread loads, B denote one block size and P total number of pixels in one subregion, respectively. Then we use a group of $S_1 = \lceil \frac{P}{KB} \rceil$ blocks for reduction of each subregion. If P is not divisible by KB, then the last block in each group is not fully utilized, and processes only $K - 1$ values because there are no more pixels in the subregion.

Once each thread loads its data, it is possible to reduce them fast within each block. We use warp–level shuffle instructions, which allows the threads in one warp to exchange data in registers. Figure 2.2 shows how it is possible to get sum of values in 8 threads. It can be expanded to size of warp, i.e. 32 threads. To reduce the whole block, we use the warp reduction in two steps:

1. We perform the reduction within each warp and save the values into shared memory.

2. One warp loads the values from the shared memory and reduces them using warp–level reduction again

Maximum number of threads in one block is 1024 in CUDA and warp size is 32 so we get at most $1024/32 = 32$ values in the first step. That means one warp is enough to reduce them into one resulting value in the second step.

After the reduction within each block, we need to update the resulting values in global memory. If S_1 is equal to 1, thus every subregion is processed by exactly one block, then there is no synchronization needed between the blocks. Otherwise, we can use *atomic add*. The solution is feasible for us, since it is safe to assume that not many blocks access the same value. The worst case size of one subregion is 250×250 which is the total of 62500 pixels. For $K = 1$ and $B = 1024$ (there is no reason not to use the maximal possible

value — no register or shared memory pressure), then 63 blocks compute the sum of one subregion.

Higher K helps utilize the threads better, because they manage to do more work before the reduction phase. For $K = 1$, half of the threads do (almost) nothing useful in the whole reduction: each of them only loads one value in the very beginning and immediately passes it to another thread (using shuffle instruction) which does the actual sum. On the other side, setting N and B such that only one block processes one subregion can lead to not being able to utilize whole gpu, if there are not enough subregions.

2.3 Arg max computation

The cross–correlated data are further processed to find the position of maximum for each subregion by using parallel reduction. The computation is very similar to the sums reduction explained in section 2.2 we use warp–level shuffle instructions and shared memory to reduce the values loaded to each block. The difference compared to sum reduction is that we need to find the maximum and its position at the same time. So throughout the algorithm, we operate on the pair of value and its position. Every time we compare two values and choose the higher one, we propagate its position as well.

Otherwise the reduction within one block is analogous to the sum reduction. In each step we shuffle down the current value and its position (that means two shuffles per one step). Then we compare the received and current value and save the greater one with its position for the next step. We need five these steps to reduce one warp, then each warp saves its result to the shared memory. Finally, one warp reduces the items in the shared memory to get the result of one block reduction.

In the sums reduction kernel, we used built-in atomic add to update the global memory with the result of block reduction. No such function exists for arg max, since we need to atomically compare two values and then update the position as well. There are several ways how do the update without the atomic operation, such as calling a second kernel that reduces the results of the first, or using only one block per subregion. However we have chosen the following one using atomic operation, since it offers a wider variety of parameters and there are never many thread blocks trying to update the same values.

For floats, we can use the atomic compare and swap (CAS) instruction to update the global memory atomically. There are not many blocks that would compete over the same piece of memory, so it is a viable solution. However, it can only be done when we compute the cross–correlation in single precision

floating point type, since modern GPUs only support atomic CAS for 8 bytes. A double precision number is 8 bytes long itself, and we need to update the pair of value and its position.

That is why we use a different trick for the double implementation. We again use the CAS, but this time we only update the positions using the instruction. In the loop of CAS, we compare the values by accessing the value in the main buffer.

The different approaches are compared in chapter 3.

2.4 Subregions preprocessing

Once we have described the most important parts of the algorithm, we explain how they are used together. In the following section, we outline the preprocessing that is needed before we can cross–correlate the subregions. It incorporates several steps:

1. Load an image of pattern from disk
2. Transfer the image from host memory to GPU memory
3. Extract the subregions of interest from the image
4. Compute the sums of subregions
5. Use the sums to compute a mean for each subregion and subtract it

In the end, all the image subregions specified by the user are organized one after another in the output buffer on the GPU. In case of cross–correlation implemented by FFT, the subregions are organized in a 4 times larger buffer with zero padding, so that it can be used by the forward Fourier transformation (see section 2.1.4).

It is not immediately clear, in what order we should perform these steps and which ones should be executed on the GPU. Naturally, load of the image from disk must come first and computation of sums must come before they are subtracted. Also, the image has to be loaded from the disk, while the rest can be implemented on GPU (the transfer step is of course special in this regard).

First, we decide whether we transfer the whole image and then extract subregions on the GPU, or extract using CPU and transfer only the subregions. If there are only few subregions, the latter approach may transfer less data, since some areas of the pattern are never used. On the other hand, if there are many subregions, they are likely to overlap, which means we could

transfer more data than needed. That said, the difference of amount of data transferred does not really matter, since empirical testing showed the time needed to copy the data from host memory to GPU is marginal compared to the rest of the algorithm. However what matters is, that we can utilize GPU for subregion extraction when we, so we have chosen that approach.

The most straightforward GPU implementation of the preprocessing is to write three kernels which first extract the subregions from the image, then compute their sums and finally subtract the means them. We can optimize it by merging the first and the final step — we first compute the sums by reading the subregions directly from the image of pattern. Then, in the second kernel called *prepare kernel*, we load the pixels of subregions from the image again, compute the mean of respective subregion and subtract it from each pixel. Finally, we write it to proper place in the output buffer. This approach saves us the overhead of starting one kernel

The output of the kernel are the normalized subregions stored in one buffer that is prepared to be cross–correlated. In case of using cross–correlation implementation based on the definition, the subregions are stored continuously side by side. In case of the FFT implementation, the subregions are stored in a continuous buffer of matrices with size $2W_s \times 2H_s$ (one subregion has the size $W_s \times H_s$), each subregion taking the left upper part of the matrix. The rest of the matrices are used for zero padding.

If some subregions are overlapping, we copy some of the data twice, but then we subtract different means from them in general, so the result does not have any duplicities.

The load of the image is implemented using the C library libTIFF [10]. Although TIFF format offers more options for encoding pictures and even compression, the testing images we received (that were produced by an EBSD camera) use simple layout without any compression which is preferable from the performance point of view. The specific testing images are divided into strips of 4 rows, which is roughly 7 kilobytes. Within the strip, the pixels are organized in row-major order, each pixel is represented by an unsigned 16 bit integer. Thus, we are able to load the strips one by one to the host memory without any major performance penalty.

The implementation of kernel that computes sums is described in section 2.2. The implementation of the prepare kernel is fairly straightforward, we assign one thread to each pixel in each subregion. The threads compute the position of their pixel in the pattern using the buffer with the positions of the subregions, then compute mean using the sums buffer, subtract it and finally write the value to the output buffer.

2.5 Cross–correlation result processing

After the cross–correlation is finished, we analyze the result — we find where is the maximum of each subregion using parallel reduction, which is explained in section 2.3. The result is an array of coordinates of the maximum values (we are not interested in the values themselves). Then, we need to process the square shaped neighborhood of each maximum to fit the values with a quadratic function and get its maximum.

The amount of data processed in this stage of computation is much smaller compared to the rest of the algorithm. In the worst case, i.e. the biggest size of neighborhood $F \times F = 9 \times 9$ that we consider and the smallest size of subregion $W_s \times H_s = 5 \times 50$, the neighborhoods are 30 times smaller than whole subregions. In addition, the offset finalization is not particularly computationally demanding and we empirically measured that in the worst case it takes less than 1% of the whole algorithm run time. Moreover, it is possible to perform the stage on the CPU in parallel with GPU computation, thus removing its impact on the total time whatsoever, as explained below in section 2.8. Those are the reasons we decided it is pointless to implement it for GPU. However in case the range of considered parameters change in the future, the addition of GPU implementation would not be a problem.

2.5.1 Extract neighbors kernel

Once we compute the position of cross–correlation maximum for each subregion, we need to transfer the maxima positions and their neighborhoods from GPU to main memory so that the CPU can finalize and output the offsets. The maxima are already ready to transfer as a result of the arg max kernel, but we need to start another kernel to copy the neighborhoods from the cross–correlation into its own buffer, so we do not have to transfer the whole cross–correlation.

Recall that F denotes the size of the square neighborhood. Given a maximum $[x_m, y_m]$, its neighborhood are the points $\{x_m - \frac{F-1}{2}, \cdots, x_m, \cdots, x_m + \frac{F-1}{2}\} \times \{y_m - \frac{F-1}{2}, \cdots, y_m, \cdots, y_m + \frac{F-1}{2}\}$ of cross–correlation. F is always an odd number, so that the maximum is in the middle of the neighborhood. So the output buffer has size $F^2 \cdot S$, because each neighborhood has $F \cdot F$ points and there are S subregions.

In the kernel, we assign one thread to each point of the neighborhood. Each thread computes the address of its point and copies it from the cross–correlation buffer to the neighborhoods buffer.

This kernel works with very small amount of data compared to other kernels, since F is expected to be less than 10. Therefore the running time

of this kernel is marginal and does not offer much space for improving overall running time.

2.5.2 Offsets finalization

So when we have the positions of maxima and their neighborhoods, we transfer them to the host memory, since the finalization of the offsets takes place on the CPU. Next step is to fit a quadratic function to each neighborhood. As explained in section 1.3, we have precomputed the matrix $(A^T A)^{-1} A^T$ for each considered $F \in \{3, 5, 7, 9\}$. Then we multiply the neighborhood as a vector with this precomputed matrix using standard matrix multiplication. That results in 6 coefficients of the quadratic function.

Then we use the formulas from section 1.3.3 to compute the maximum of the continuous function given by the 6 coefficients, which gives us offset of the maximum from the middle of each neighborhood. In the final step, we combine the location of neighborhoods transfered from the GPU with the relative offsets and write out the result.

2.6 Data flow

Figure 2.3 summarizes the whole algorithm and shows all the used kernels and buffers (apart from additional buffers that are used to compute the cross–correlation).

Table 2.2 shows all buffers used on GPU with their sizes. The kernels that compute sums and normalize the subregions both work with $W_r \cdot H_r \cdot S$ items (i.e. all subregions). The cross–correlation has nearly four times as big output, which is then processed by the arg max kernel. So the cross–correlation and the arg max kernel work with the largest amount of data, while the buffers that are transferred to the CPU memory are quite small.

The table 2.2 also shows the data types stored in the buffers. The input image consists of unsigned 16 bit integers, since that is the format of our testing images. Four byte unsigned integer is then enough to hold the sums of its subregions of expected size. The biggest number that the 16-bit unsigned integer is $2^{16} - 1$. That means we can sum at least $\frac{2^{32}}{2^{16}} = 2^{16}$ pixels, before the variable overflows. The square root of 2^{16} is $2^8 = 256$, which is bigger than the considered range of subregion sizes. Should the range increase in the future, the implementation has to be changed to reflect that.

Once the means are subtracted, the pixels are not integers in general, so we need to use a floating point type. We implemented the algorithm for both single or double floating point types, since the single precision version is

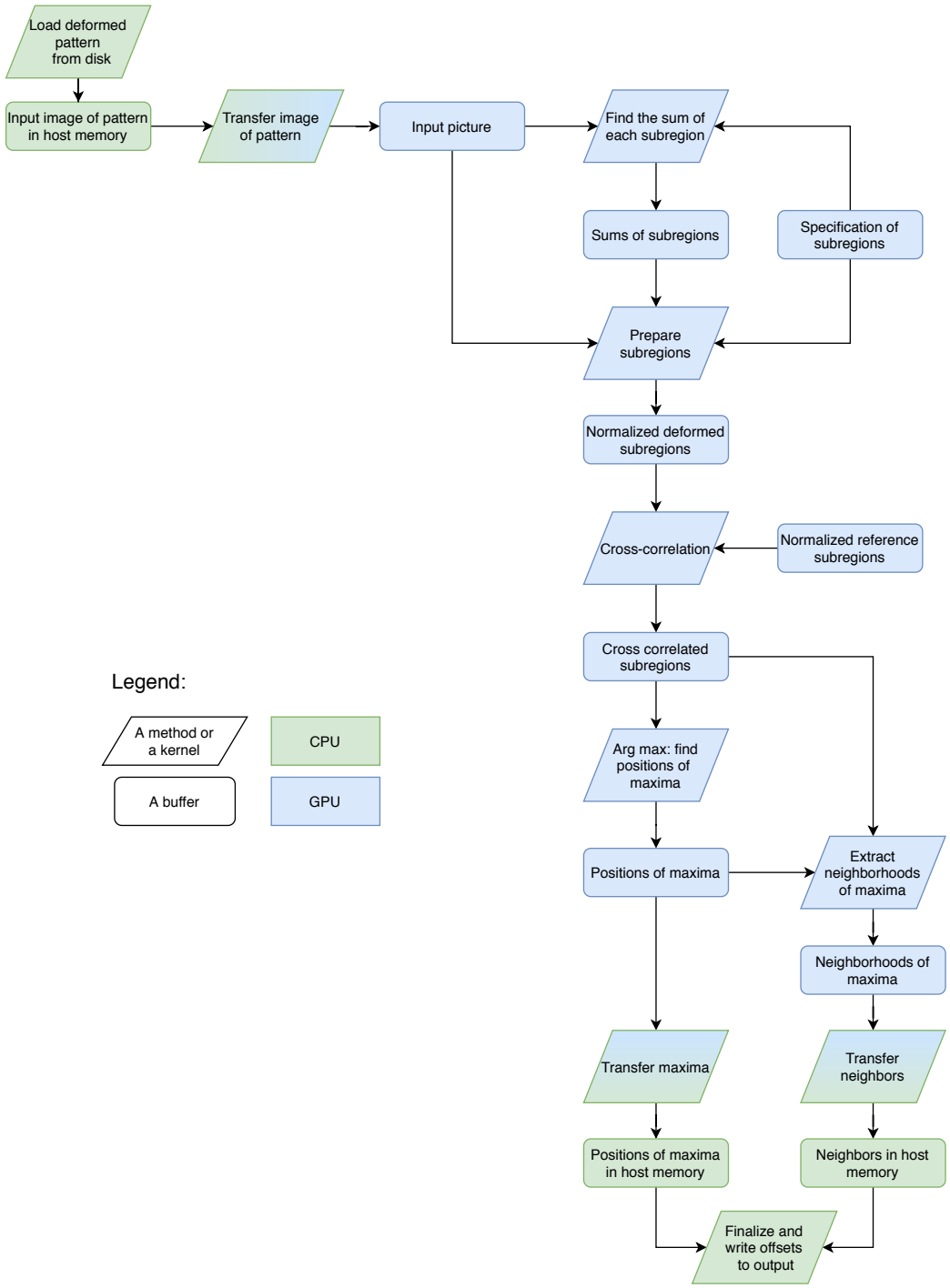


Figure 2.3: Processing of one deformed pattern.

Buffer	Type	Number of elements
Input picture	uint16	$W_p \cdot H_p$
Sums of subregions	uint32	S
Positions of subregions	uint32 pair	S
Normalized deformed subregions	float/double	$W_r \cdot H_r \cdot S$
Normalized reference subregions	float/double	$W_r \cdot H_r \cdot S$
Cross–correlated subregions	float/double	$(2W_r - 1) \cdot (2H_r - 1) \cdot S$
Positions of maxima	uint32 pair	S
Neighborhoods of maxima	float/double	$F^2 \cdot S$

Table 2.2: Summary of GPU buffers with their data types and size.

substantially faster, but it is up to the user and the interpretation in physics whether its precision is sufficient. The result of the algorithm is further processed and the resulting rotation of the pattern is expressed in quite small numbers (in the order of 10^{-4}), thus small imprecisions introduced by smaller floating point types might matter. That is also why we did not implement the algorithm for half float, but if the 4 byte float proves sufficient, half float may be worth implementing in the future.

2.7 Image batch processing

The algorithm we explained so far in this chapter processes the patterns one by one, never processing subregions from different images in parallel. However that can be changed to reduce the overhead of launching kernels and ensure that the GPU is fully utilized even when we process a small number of subregions in each image. We implemented a parameter that specifies number of patterns that are processed in batch in parallel. In this section, we describe how it changes the already described implementation.

The load of images from disk does not change, we just need to load several images in a row. The sum and prepare kernels have to be slightly changed, because they have to use the specification of subregion locations multiple times — once for each image.

The complex matrix multiplication kernel, that works with the reference subregions, benefit from processing more images at once, since each of them is multiplied with the same reference. Each thread is now able to load the value from reference image only once and then use it multiple times.

2.8 Task parallelization

Since some parts of the algorithm are executed by CPU and some by GPU, it is possible to parallelize them. There are 3 parts separated by data transfers between the main and the GPU memory: CPU first loads a pattern, GPU processes it, and then CPU does the finalization of results. It is desirable that we parallelize those tasks, so that GPU is fully utilized and never waits for the CPU. See also the color distinction in fig. 2.3.

We use a three stage pipeline to accomplish the parallelization. The first stage loads the TIFF images of patterns. The second one processes them on the GPU and the third CPU one finalizes the results and writes them to the output. All of the stages run in parallel, using two queues to synchronize.

Since all the stages run in parallel, the whole pipeline will have the performance of the slowest stage. As argued in section 2.5.2, the offsets finalization phase works with little data compared to the rest of the algorithm and in our empirical testing it was never a bottleneck. The two remaining stages are a bottleneck depending on the parameters.

Duration of the stages depend on different things. The GPU stage duration depends on the size and number of subregions and obviously the performance of the GPU chip. On the other hand, the loading stage depends on the size of loaded images (but not on the subregions that we compare), performance of the disk and also the CPU. So for high number of big subregions per image, the GPU will be most likely the bottleneck, while for high–end GPUs and smaller subregions, the disk throughput is the bottleneck.

Moreover, we found a scenario in which one CPU thread is unable to fully utilize the disk. The TIFF format organizes the data in chunks, which means that CPU needs to do some work between loads of the chunks to the memory. To help fully utilize the disk, we implemented the possibility to load several images in parallel in more threads. Overall design of the pipeline stays the same, just the first stage is multiplied. The threads still add the loaded images to the same queue.

We also partially take advantage of the fact that GPUs are able to run a kernel, copy data from and to memory at the same time. Actually, all transfers run in parallel with kernel execution. The host–device copy of input takes place right after the CPU loads the pattern from disk. Similarly, once the maxima and neighborhoods are prepared on the GPU, we start asynchronous transfer of the results and GPU starts computing the next pattern immediately. Unfortunately, this optimization does not affect performance much, since the copying between GPU and CPU memory takes only a marginal fraction of overall time.

3. Benchmarks

In this chapter, we measure the performance of our implementation. We quantify the impact of different input size and algorithm parameters. In the end, we compare our implementation with an original python implementation of the algorithm.

All the data was measured using an NVIDIA RTX 3080 graphics card with Intel i5 6600K processor. We used an M.2 SSD AData SX8200 Pro.

To measure the values presented in this chapter, we used a data set provided by the Department of Physics of Material in Charles University in Prague. It contains roughly 15000 electron backscatter patterns — each of them is a TIFF picture with resolution 873×873 with color depth of 16 bits. One image has 1.4 megabytes and the whole dataset has 22.3 gigabytes. Most of the testing was performed using only a small subset of the data, since it is needlessly big for most measurements. The exception is the overall comparison to the python implementation and image loading tests for which we needed to use data bigger than the cache of disk.

To measure the performance of individual parts of algorithm we edited the implementation to measure that the respective part takes to complete and write out the average after all the input images (a subset of the 15000 patterns) are processed. So the durations of individual parts are measured directly during running of the whole algorithm. Moreover, we started the whole application 8 times and again took the average. In order to further avoid results variation, we unplugged the computer from the Internet and reduced number of running processes to minimum. This all resulted in negligible performance differences in the separate runs. All the following results depict the averages.

Besides the patterns, the most important input to the algorithm is the description of subregions — their amount and size of each of them. The actual location of subregions does not have an impact on performance. Throughout the chapter, S denotes the number of subregions. Also, we measure only square subregions, because it removes unnecessary complexity from the benchmarks and it is used with square subregions in practice. Moreover, in this whole chapter, the size of a subregion is half of length of the rectangle side. So for size of subregion A, the actual subregion is $2A$ wide and $2A$ high. The reason for this is that the subregion positions determine its middle and it is easier to specify the size as the distance between the middle and sides of the squares.

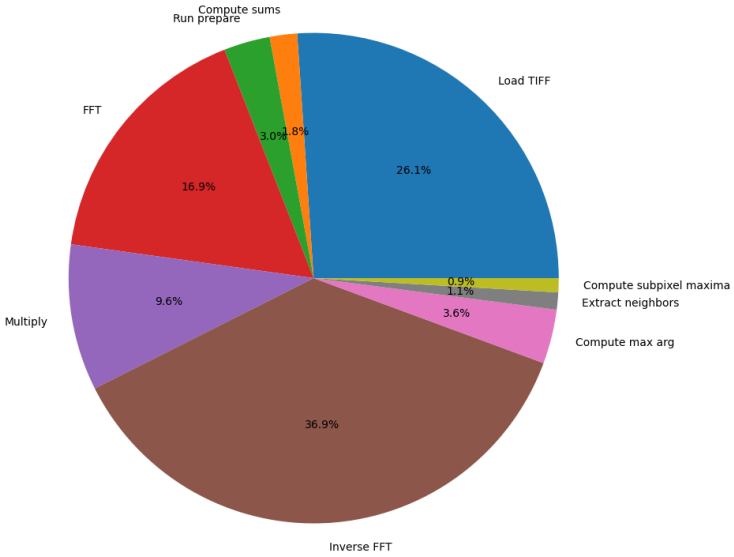


Figure 3.1: Relative performance of individual parts of the algorithm. The data was measured from a run of the algorithm with a configuration of 90 subregions with size 120×120.

3.1 Relative performance of individual parts

We begin with an overview of all the kernels used in the implementation. Figure 3.1 shows how much time the implementation spends in individual kernels. We can see, that the most demanding parts are loading of the TIFF and the cross–correlation, which is computed using the multiply kernel and the Fourier transforms.

The figure is mentioned here only to get general idea about the performance. It captures relative performance of the kernels for 90 subregions with size 60, which is roughly in the middle of the range of expected values. Different parts depend on different parameters, so for some other input size, the relative performance would look dissimilar. The load TIFF subroutine depends only on the size of the TIFF, while the rest of the algorithm depends on the number of subregion. So, for instance, it is possible to find a configuration that makes load TIFF the longest part.

3.2 Cross–correlation

First, we measure the performance of cross–correlation when computed using Fourier transform. We show the dependence between running time and size and number of subregions. Since the cross–correlation is computed using

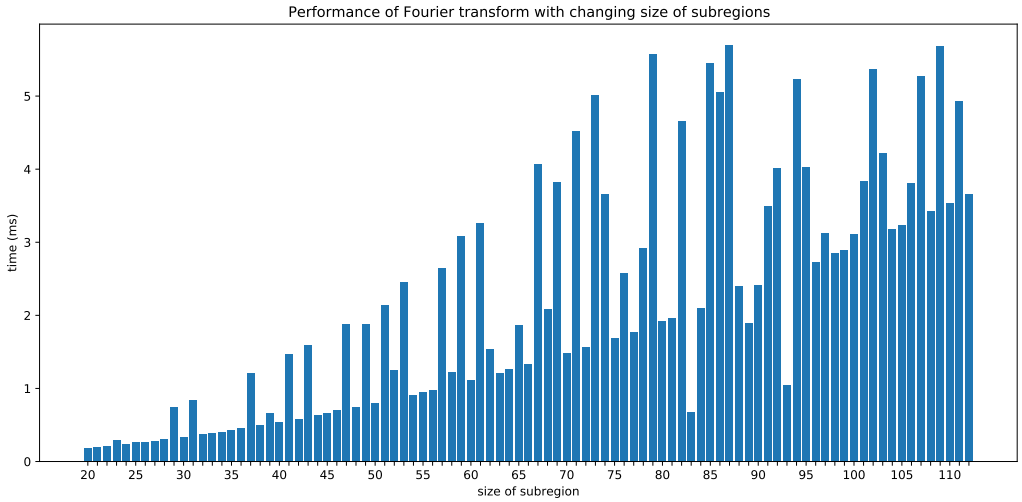


Figure 3.2: Performance of Fourier transform with changing size of subregions. It was measured from a run of the algorithm with a configuration of 110 subregions.

three parts, we analyze them one by one. Then we compare the FFT and definition–based implementations.

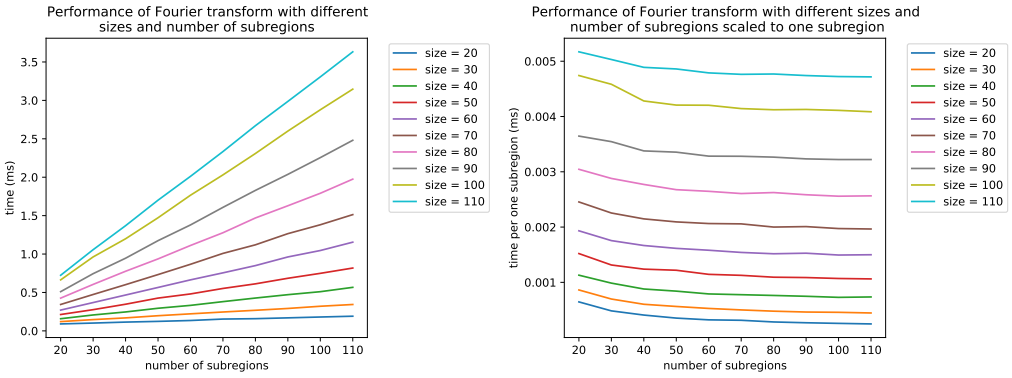
3.2.1 Fourier transforms

In the fig. 3.2, we can see how the forward Fourier transform executed by the cuFFT library depends on the size of subregions. We can see that the computation time increases with the size of the subregions, following the $\mathcal{O}(n \log n)$ time complexity of FFT, where n is number of pixels. More precisely, the complexity is $\mathcal{O}(A^2 \log A)$, if A denotes the length of one side of the square subregion.

However, there are quite a few outliers — sizes, for which the function does not perform very well. That can be explained by the following sentence cited from the cuFFT documentation: "Algorithms highly optimized for input sizes that can be written in the form $2^a 3^b 5^c 7^d$". It means that all sizes, whose factorization contains a prime greater than 7 perform poorly. Choosing a bigger size with nice factorization results in better performance.

We can also see, that sizes 83 and 93 perform surprisingly well. We have measured this fact consistently. The reason behind such behavior is probably specific for cuFFT implementation.

The dependence of inverse Fourier transform on size of subregions looks very similar to the fig. 3.2 for the same reasons.



(a) (b)

Figure 3.3: Performance of Fourier transform with changing size and number of subregions.

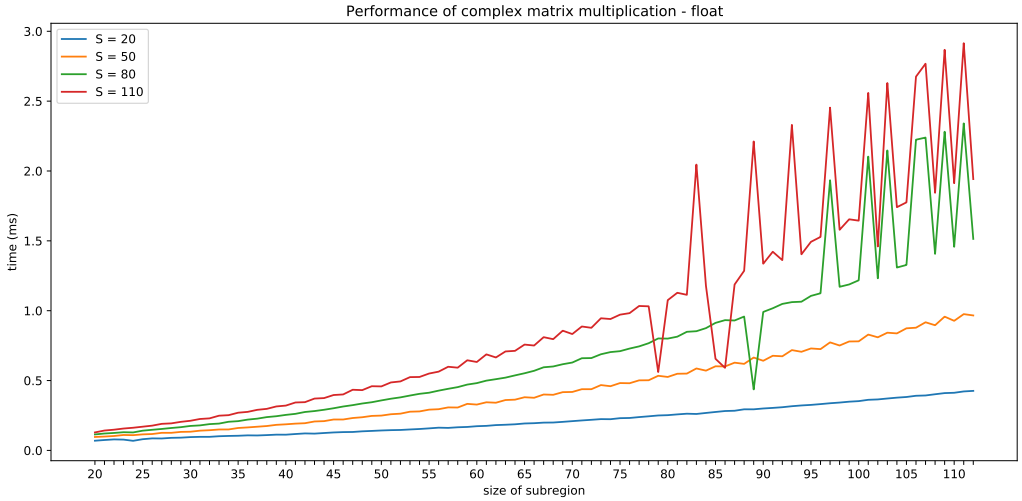
Figure 3.3 shows how the the Fourier transform performs with different sizes and number of subregions. Figure 3.3a shows absolute time it takes to compute the Fourier transform. As expected, the time scales roughly linearly with the number of subregions in each image.

Figure 3.3b shows the same dependence, but the times are scaled with respect to number of subregions, so the y axis shows the average time needed to compute the Fourier transform of one subregion of respectable size. We can see that regardless of the size of subregions the library needs less time per subregion with increasing number of subregions. It is most likely caused by overhead of each CUDA kernel call — with more subregions, the same overhead is spread over more of them, thus the computation of each subregion is cheaper. That is also the reason why we introduced the batch parameter — to process more subregions in each run of kernel. We evaluate the impact of the batch parameter to the whole algorithm in section 3.5.

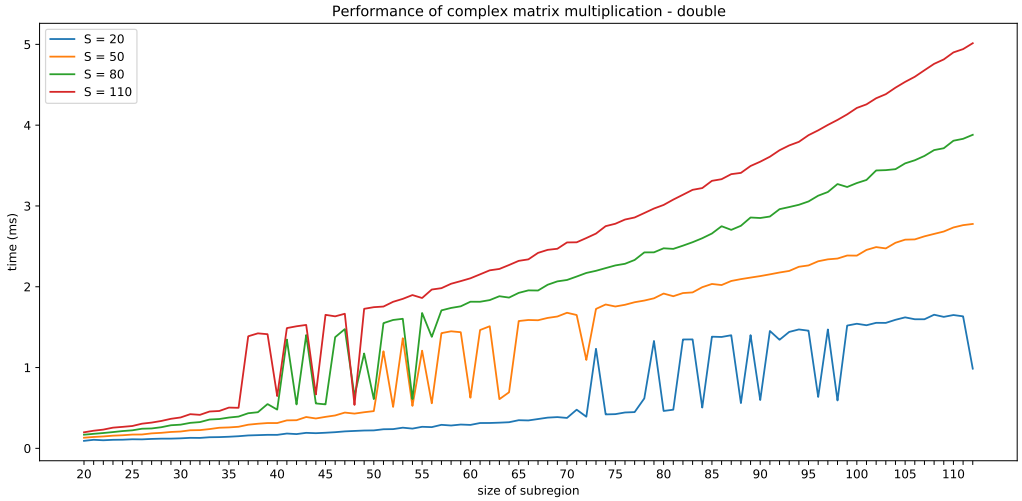
3.2.2 The complex matrix multiplication

Another step in the cross–correlation computation is the multiplication of two complex matrices. In theory, the complexity of the multiplication is $\mathcal{O}(A^2S)$, where A denotes the length of one side of the square subregion and S denotes their amount. Figure 3.4 shows the measured time for different sizes and number of subregions. We can see the quadratic trend with respect to subregion size and linear with respect to their count.

However, in some cases, there seem to be a threshold over which the kernel performs worse. For float precision (fig. 3.4a), the thresholds are higher than



(a) Single float precision



(b) Double float precision

Figure 3.4: Performance of complex matrix multiplication with changing size and number of subregions.

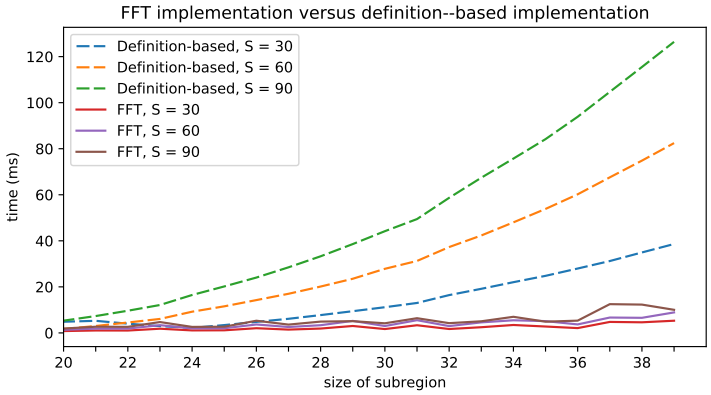


Figure 3.5: Comparison between the FFT and definition–based implementations of cross–correlation.

for double precision (fig. 3.4b) and we do not see a consistent increase of the time for the measured sizes, only individual spikes. Unfortunately, we cannot reliably determine the relation between the threshold and algorithm parameters. It seems that the data exceed some kind of cache, but we observe worse performance for such big sizes that the processed subregions are already bigger than tens of megabytes. The plots look similar for different batch sizes too.

Moreover, we tried to measure the performance of the multiplication kernel individually and the observed worse performance was not present — we measured clean quadratic dependency on size of subregions. That implies it is a result of some interference with the cuFFT forward Fourier transform. That would also explain the spikes, since cuFFT behaves differently for sizes with primes in their factorization. In any case, the performance difference is roughly 1ms, which is up to 10% of total running time of whole offsets computation.

3.2.3 FFT implementation versus definition–based implementation

Recall that we described two implementations of cross–correlation: first one based on the Fourier transform and second one based on the definition. Figure 3.5 shows comparison between them — for the FFT version, it shows the sum of average times of the Fourier transforms and complex multiplication. For the definition–based implementation, it shows the average run time of respective kernel. They are roughly the same for very small subregion sizes, but for sizes bigger than 25 the FFT implementation clearly wins. That is

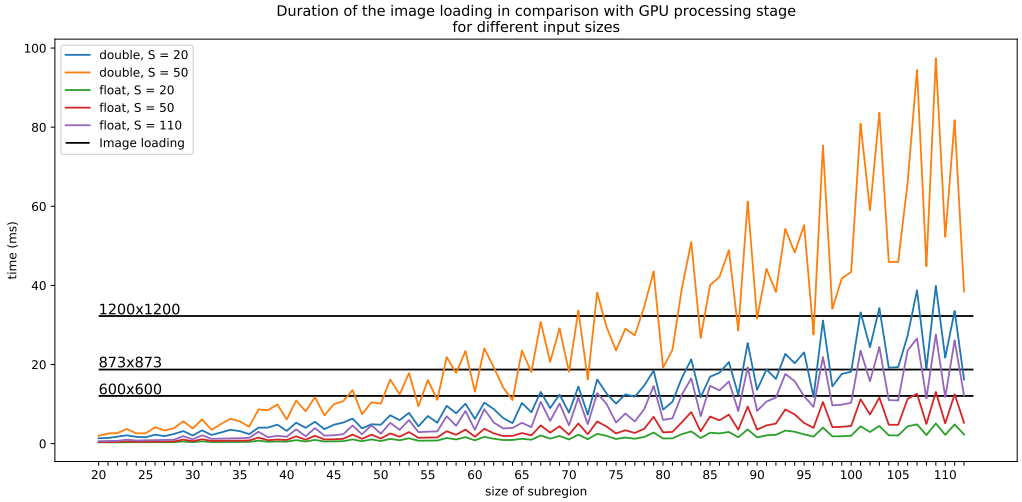


Figure 3.6

the minimal size of subregions that we consider useful for practical application. The conclusion is that the definition–based implementation is not useful for our application and it is always better to use the FFT implementation. Therefore we use only the FFT implementation in the rest of this chapter.

The rest of the implemented kernels take up less than 10% of total GPU processing time. Therefore improving the performance of any of them by significant margin does not have almost any impact on the resulting performance. Thus, we rather move on to benchmarking of pattern loading, which does take up significant time.

3.3 Pipeline benchmarks

In the following section, we benchmark the pipeline parallelization of pattern loading and GPU processing described in section 2.8. The duration if the image load is simply a linear in number of pixels being loaded, or quadratic function of the width of the image. We show its performance in comparison with the GPU processing time in fig. 3.6. There are three black lines which represent the duration of image loading for three different sizes of patterns. They are horizontal, since pattern load does not depend on size of subregions. It was measured using only one CPU thread on a large amount of data, so we ensured that it was not preloaded in disk cache.

S\size	20	50	80	110
20	11710.0	2362.8	1213.5	471.4
50	6469.3	1041.5	534.9	195.4
80	4856.7	684.6	341.7	123.3
110	3765.3	515.1	251.4	90.0

Table 3.1: Disk throughput needed to fully utilize the tested GPU for different sizes and number of subregions when processing in double precision. All the values are in megabytes per second.

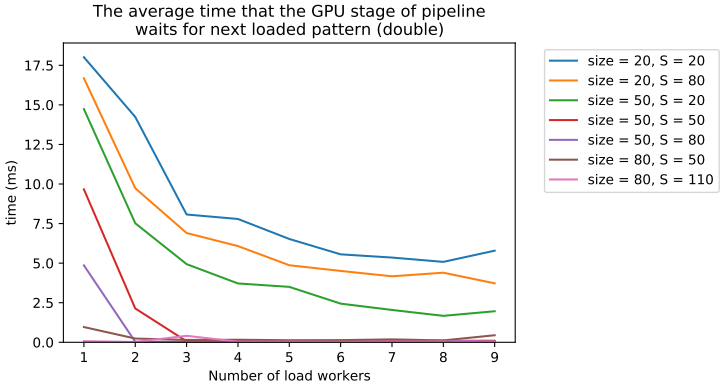
Additionally, there are lines that represent the duration of the GPU processing for different number of subregions and floating precision. The GPU processing incorporates all the steps that are implemented for GPU including cross–correlation. For different sizes, we can see the spikes for sizes with primes in their factorization that are explained in section 3.2.1. Otherwise, the time is increasing with increasing size and number of subregions, as expected.

In the fig. 3.6, we can also see which stage of the pipeline is the bottleneck for different parameters. Since the pattern loading and GPU processing are executed simultaneously, whichever of them takes longer is the bottleneck. It is clear that for considerable number of configurations, the amount of data that one thread can load from the disk is insufficient to fully utilize the GPU in the testing computer. The throughput of the image loading with size 873×873 was roughly 500 MB/s, which means we did not fully utilize the used M.2 SSD either.

3.3.1 Impact of using more load image threads

To utilize the system better, we use several threads to load more patterns simultaneously. The goal is to load the patterns as fast as possible, so that the GPU waits as little as possible. Figure 3.7a shows the average time that the GPU stage waits for the next pattern. We used input images with resolution 873×873. We can see that for bigger subregions and bigger number of them, the GPU thread waits less in general. Also, more load workers generally decrease the waiting. And with some of the configurations, more load workers eliminate waiting completely, while the ones with less GPU work cannot fully utilize the GPU even for bigger number of load workers.

To explain the reason why we see two different types of behavior we analyze the throughput of testing system components. Table 3.1 shows the amount of images that the testing GPU is able to process each second in



(a)

(b)

Figure 3.7: The average time it takes to load image and the average time the GPU stage waits for loaded patterns, all for different number of load threads.

megabytes. To calculate it, we divided the size of one batch of patterns by the measured time it takes to process it by the GPU, which gives us the throughput of loading images that is needed to fully utilize the GPU. To put those numbers in context, a hard disks have read speed of around 100 MB/s, common read speed of SATA SSDs is 500 MB/s and M.2 SSDs can go up to 3500 MB/s. On the testing SSD, we measured read speed of roughly 1500 MB/s when the data was not present in the disk cache.

This information combined with the table 3.1 explains fig. 3.7a. For configurations with small number of small subregions (all the configurations with size of 20 and [size = 50, S = 20]), the GPU can process more data than can be loaded from the disk per second. Nevertheless, more load workers allow for better utilization of the disk. For bigger subregions, several load workers are needed to achieve larger disk throughput than the GPU can process and from that point, the GPU does not wait for the image loading anymore.

Figure 3.7b shows the average time that one thread needs to load the image. We can see that for disk–constrained configurations the time each thread needs to load the image increases, as the disk is more loaded. However, at the same time, together they manage to load the images faster. For the GPU–constrained configurations, the load time depends on number of threads needed to utilize the GPU.

All the measured values are, of course, specific to the testing machine. At the same time, it shows that for some configurations, having a fast SSD can be even more important than having a powerful GPU.

3.4 Float versus double

Figure 3.8 shows how much time is needed to process the subregions by the GPU. We can see that the plot is very influenced by the performance of the cuFFT library. For sizes with big primes in their factorization, we can see the spikes as explained in the section 3.2.1. The algorithm performs better in single precision, float version being faster for each size, regardless of number of subregions. More precisely, we measured that the single precision configuration is $4 - 11$ times faster depending on the configuration, 7.27 times on average.

In the previous section, we mentioned table 3.1 that contains the disk throughput required to fully utilize our testing GPU for double precision. Table 3.2 contains the same for single precision floats. We can see, that for most of the configurations, our testing SSD (with the throughput of 1500 MB/s) is not enough to utilize the GPU. That underlines the necessity of

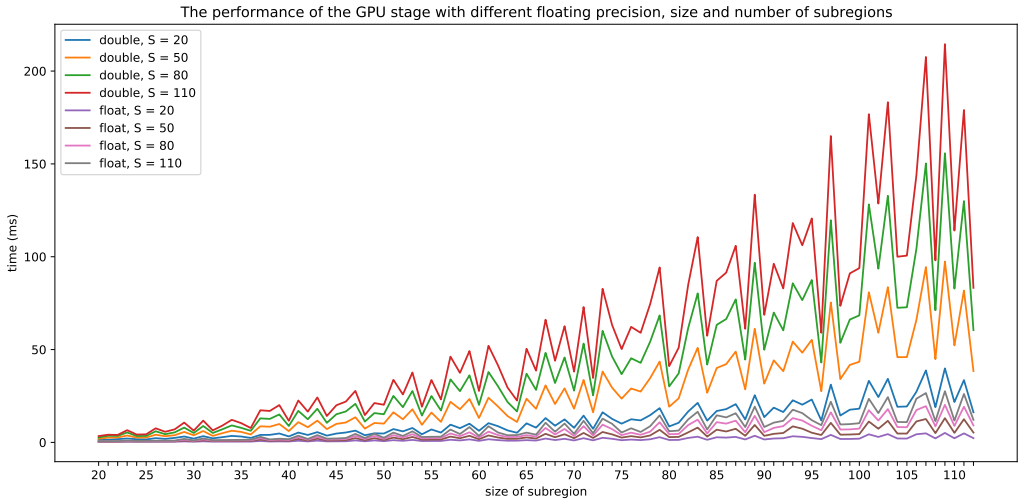


Figure 3.8

S\size	20	50	80	110
20	30650.2	15497.3	7810.9	4585.8
50	25519.6	7929.9	3570.0	1984.7
80	20452.4	5378.6	2311.6	1148.5
110	16735.2	4049.2	1699.2	865.1

Table 3.2: Disk throughput needed to fully utilize the tested GPU for different sizes and number of subregions when processing in single floating precision. All the values are in megabytes per second.

mean	0.005%
95-th percentile	0.004%
99-th percentile	0.049%
99.9-th percentile	0.614%
99.99-th percentile	5.578%
max	12.757%

Table 3.3: The error of single floating point precision compared to double precision

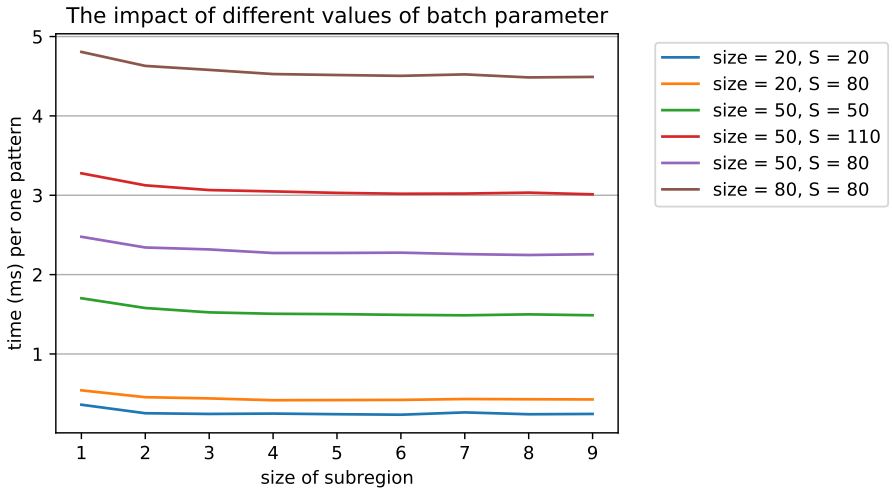


Figure 3.9: The impact of the batch parameter on the performance of GPU part

fast SSD, should the user of the implementation choose the single precision.

We also measured how much the results between different precisions varied. On average, the shifts differed by 0.005%. Table 3.3 shows more statical characteristics of the error. Whether the single precision is sufficient depends on physical interpretation.

3.5 Batch parameter

Figure 3.9 shows the time needed to process one pattern with different values of the batch parameter for selected subregion configuration. With increasing size of batch, the absolute time needed to process it increases (almost) linearly, but we also process less batches. That is why we show the average time needed to process the subregions of one pattern. We can see, that the

S\size	20		50		80		110	
20	34.9%	5	20.7%	6	13.7%	8	7.7%	7
50	26.1%	7	12.6%	6	8.5%	6	4.7%	8
80	23.0%	5	9.3%	7	6.7%	7	3.7%	7
110	28.8%	8	8.1%	8	5.5%	8	3.0%	8

Table 3.4: The comparison between the best batch value compared to size of batch equal to one. Both relative improvement and best value are shown.

time improves with the higher values of the parameter.

Table 3.4 shows the best performing value of batch parameter for selected configurations and relative improvement compared to the size of batch equal to one. We can see, that the best improvements are achieved for smaller number of smaller subregions. The possible reason is that when processing of one pattern takes so little time (less than 0.5 millisecond), the overhead of communicating with the GPU and starting kernels takes up significant portion of that time. Also, it is possible that the amount of data is so small that it is unable to utilize whole GPU.

We have not measured any impact of the batch parameter to loading of the pictures from the disk.

3.6 Speedup compared to reference python implementation

In the following section, we measure the speedup of the GPU implementation compared to a python implementation provided by Jozef Veselý from Department of Physics of Material in Charles University in Prague. At its core, the implementation relies on FFT implemented in python library scipy, which is able to utilize all CPU cores. Besides that, is uses numpy functions like `mean` and `argmax` to perform the additional processing, which offer an optimized single–threaded implementation. Thus, the most expensive part uses multi–threaded implementation, but the rest uses only one thread.

Table 3.5 shows the comparison between the python and the GPU implementation. We processed the whole testing dataset of over 15000 patterns to get those durations. Many durations in the table are around the 17 second mark. Those are the configurations that are bottlenecked by the disk throughput. Otherwise, we see roughly 30x to 40x speedup for computing with double precision and up to 270x speedup for large size and number of subregions in floating point precision. All the other configurations in single

size	number of subregions	python time (s)	GPU, double time (s)	GPU, double speedup	GPU, float time (s)	GPU, float speedup
20	20	328.5	17.4	18.9x	18.1	18.2x
20	110	922.7	16.8	55.0x	18.1	50.9x
50	50	775.5	24.6	31.6x	17.3	44.9x
50	80	1124.0	36.4	30.8x	17.4	64.5x
80	50	1840.2	46.9	39.2x	17.5	105.2x
110	20	1714.4	51.4	33.3x	16.9	101.4x
110	110	8181.1	254.9	32.1x	29.9	273.5x

Table 3.5: The comparison between python and GPU implementation for different floating precision and subregions description

precision are bottlenecked by the disk.

In the introduction, we mentioned that modern EBSD cameras can take up to hundreds of pictures per second. When we convert the times from table 3.5 to processed pictures per second, we get that the GPU implementation can process from 65 to 925 patterns per second depending on the parameters. Thus, the performance of our GPU implementation and the speed of modern EBSD cameras are of the same order of magnitude.

Conclusion

In this thesis we have described the processing of the electron backscatter diffraction patterns (chapter 1), including the cross–correlation and how is it computed using the Fourier transform(section 1.2.1). We have shown how is the cross–correlation subsequently processed to obtain estimation of position of the maximum with subpixel precision (section 1.3). The algorithm is summarized in pseudocode in section 1.4.

Next, we have analyzed individual parts of the algorithm and described their CUDA implementations (chapter 1). We use the cuFFT library to compute the cross–correlation (fig. 2.1) and leverage the parallel reduction to implement kernels that find the sum of pixels in each subregion (section 2.2) and the positions of their maxima (section 2.3) Then we described how to put them together as smoothly as possible and explained that the GPU processing can run in parallel with loading of input patterns from disk section 2.8.

Finally, in chapter 3, we have measured the performance of individual parts and compared to a reference python implementation. It runs on the CPU and is even partly able utilize more cores (the cross–correlation is implemented using an optimized multi–threaded Scipy function). We have achieved the speedup of 18 – 40-times for double floating precision, which yields results identical to the reference implementation. For single floating precision, we measured speedup of 18 to 270-times. We found out that, especially the lower precision version (and the double version for small subregions), is bottlenecked by the throughput of contemporary consumer–grade M.2 SSDs (section 3.3.1). Whether the higher performance of single float version is useful is up to physical interpretation of the error introduced by computation in lower precision.

Regardless of the precision, this thesis has the potential to considerably improve the EBSD processing. It improves the performance of EBSD data analysis so it is closer to the speed of EBSD cameras, so they are in the same order of magnitude at least for some parameters (it depends both on the settings of the camera as well as the analysis). Thus, all the data that the physicists are able to measure can be processed in similar time. However, the biggest contribution of this thesis is that it makes it possible to process bigger datasets in reasonable time, opening new possibilities for researchers.

Future work

Although the implementation described in this thesis is useful and certainly improves the EBSD analysis performance, it is only one part of the whole process. In the future, we need to make sure that subsequent stages are able to efficiently process the subregion shifts produced by the program. It is possible that binary output of the implementation would be beneficial.

We can also see some possibilities to improve the performance of our solution itself. We have shown that disk throughput, or even the CPU, can limit the performance. The main goal of the thesis was the GPU implementation and we used libTIFF to load the patterns. Deeper analysis of the loading part or tighter adjustment for specific EBSD camera may result in better disk utilization.

Should the GPU prove to be the bottleneck for chosen parameters, we also see the prospect of further improvements there. One is to implement the algorithm for more GPUs. It should be pretty straightforward, since it is possible to add workers to the GPU stage of the pipeline. And if the physicists evaluate, that single float precision is enough, it might be worth to implement the whole algorithm for half (16 bit) floating precision.

Bibliography

[1] Howard Anton and Chris Rorres. *Elementary linear algebra: applications version*. John Wiley & Sons, 2013.

[2] TB Britton and Angus J Wilkinson. "High resolution electron backscatter diffraction measurements of elastic strain variations in the presence of larger lattice rotations". In: *Ultramicroscopy* 114 (2012), pp. 82–95.

[3] Wikimedia Commons. *Visual comparison of convolution, cross-correlation and autocorrelation*. 2016. URL: https://en.wikipedia.org/wiki/Cross-correlation#/media/File:Comparison_convolution_correlation.svg.

[4] NVIDIA Corporation. *2D Image And Signal Performance Primitives (NPP) Version 11.0.1.**. URL: https://docs.nvidia.com/cuda/npp/index.html (visited on 11/15/2020).

[5] Edax. *DigiView EBSD Camera*. URL: https://www.edax.com/products/ebsd/digiview-ebsd-camera.

[6] Edax. *Hikari EBSD Camera Series*. 2014. URL: https://www.eden-instruments.com/en/download/hikari-ebsd-camera-series/.

[7] Durgaprasad Gangodkar et al. "Efficient Variable Size Template Matching Using Fast Normalized Cross Correlation on Multicore Processors". In: *Advanced Computing, Networking and Security*. Ed. by P. Santhi Thilagam et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 218–227. ISBN: 978-3-642-29280-4.

[8] T. Idzenga et al. "Fast 2-D ultrasound strain imaging: the benefits of using a GPU". In: *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control* 61.1 (2014), pp. 207–213. DOI: 10.1109/TUFFC.2014.2893.

[9] JP Lewis. "Fast Template Matching. 1994". In: *URL http://www. scribblethink. org/Work/nvisionInterface/vi95_lewis. pdf. Cited on* (), p. 6.

[10] *LibTIFF - TIFF Library and Utilities*. 2019. URL: http://www.libtiff.org/.

[11] Yunhui Liu, Qi Zou, and Siwei Luo. "GPU Accelerated Fourier Cross Correlation Computation and Its Application in Template Matching". In: *International Conference on High Performance Networking, Computing and Communication Systems*. Springer. 2011, pp. 484–491.

[12] Justin Luitjens. *Faster Parallel Reductions on Kepler*. 2014. URL: https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/ (visited on 07/16/2020).

[13] MATLAB. *9.7.0.1190202 (R2019b)*. Natick, Massachusetts: The MathWorks Inc., 2018. URL: https://www.mathworks.com/help/signal/ref/xcorr2.html.

[14] John G Proakis and Dimitris G Manolakis. "Digital signal processing". In: *PHI Publication: New Delhi, India* (2004).

[15] Adam J Schwartz et al. *Electron backscatter diffraction in materials science*. Vol. 2. Springer, 2009.

[16] AJ Wilkinson et al. "High-resolution electron backscatter diffraction: an emerging tool for studying local deformation". In: *The Journal of Strain Analysis for Engineering Design* 45.5 (2010), pp. 365–376.

[17] Angus J Wilkinson, Graham Meaden, and David J Dingley. "High resolution mapping of strains and rotations using electron backscatter diffraction". In: *Materials Science and Technology* 22.11 (2006), pp. 1271–1278.

A. User guide

This section describes how to build the attached implementation.

The project is written in C++/CUDA and uses the CMake build system. Requirements for the build:

- git
- CMake version at least 3.9
- C++17 compiler (tested on MSVC from Visual Studio 16.8, GCC 8.3 and GCC 9)
- CUDA compute capability: 6.0

Tested on 2 platforms so far:

- Windows - Visual Studio 2019, CUDA 10.2
- Linux - GCC 8.3, CUDA 10.2

Windows

1. Open the project folder in Visual Studio. It should detect CMake project and start configuration.
2. Choose the x64-Release configuration.
3. Build → Build all (F6)
4. The resulting executable emida.exe is in `build/x64-Release/bin` folder.

You may also run `build/x64-Release/bin/emida_test.exe` to verify the build. The test has to be run from the `build/x64-Release/bin/` folder.

Linux

On linux, it is preferred to use system libtiff (it is downloaded and built from source on Windows). Alternatively, there is `TIFF_FROM_SOURCE` switch that enforces building libtiff from source, but ZLIB and LibLZMA packages are still required by the libtiff library. The following steps are an example how to build the project on Ubuntu. We assume that a compatible C++17 compiler is set as default.

1. Install the libtiff, cmake and git packages:

   ```
   apt-get install -y cmake git libtiff-dev
   ```

2. Run the following in the terminal from the project folder.

   ```
   mkdir build && cd build
   cmake ../
   cmake --build .
   ```

3. The resulting executable emida is in `build/bin` folder.

You may also run `bin/emida_test` to verify the build. The test has to be run from the `bin/` folder.

How to run

The application compares one picture of the material patern with images of deformed material. The deformed images are specified by an input file that has lines in following format:

```
<image_pos_x> <image_pos_y> <image_file_name>
```

Example:

```
0.0 0.0 test_data/deformed/DEFORMED_x0y0.tif
600.0 0.0 test_data/deformed/DEFORMED_x600y0.tif
1200.0 0.0 test_data/deformed/DEFORMED_x1200y0.tif
1800.0 0.0 test_data/deformed/DEFORMED_x1800y0.tif
300.0 519.615242270 test_data/deformed/DEFORMED_x300y519.tif
900.0 519.615242270 test_data/deformed/DEFORMED_x900y519.tif
```

Positions and size of regions that are compared in each picture are specified in a configuration file in fillowing format:

```
<size/2>\n
<roi_mid_x> <roi_mid_y>\n
<roi_mid_x> <roi_mid_y>\n
...
```

The size of each region is specified on the first line. The size is half ("radius") of the compared regions. Then, list of pairs follow, each of them specifies a middle of one subregion.

Running the executable

The executable cross-correlates parts of pictures(slices) in specified positions and then computes how much are the deformed picture's slices shifted compared to the initial picture.

```
emida -i test_data/initial/INITIAL_x0y0.tif -d test_data/def.txt \
 -b TEST_RUN/roi-cryst.txt --batchsize 7 \
 --crosspolicy fft > offsets.txt
```

Following options are mandatory:

- `-i,--initial` specifies path to the reference image
- `-d,--deformed` specifies path to file with list of the deformed pictures to process. The format of the file is described above.
- `-b,--slicepos` specifies path to file with positions of regions to be compared in each picture, as descibed above.

Optional options:

- `-s,--slicesize` overrides the size of subregions specified in the subregion description passed by the `slicepos` parameter.
- `-q,--writecoefs` In addition to offsets, output also coefficients of parabola fitting.
- `--precision` specifies the floating type to be used. Allowed values: `double`, `float`.
- `f,fitsize` specifies the size of neighbourhood that is used to fit the quadratic function to cross–correlation and find subpixel maximum. Allowed values: 3, 5, 7, 9
- `crosspolicy`, Specifies whether to use FFT to compute cross correlation. Allowed values: brute, fft.
- `batchsize`, Specifies how many files are processed in one batch in parallel.
- `loadworkers`, Specifies the number of workers that load input patterns simultaneously.
- `-a` When specified, the executable measures the duration of selected kernels and parts of processing.

Running the executable with included test data

The package contains testing data in the `test_data` folder. `test_data/initial` contains the reference, unmodified pattern. `test_data/deformed` contains the deformed patterns. Due to limited options of uploading large attachments, we attached only a subset of original testing data. `test_data/def.txt` contains list of the files in the format described above that can be loaded by the executable.

The deformed images have file names in the following format:

`DEFORMED_x<X_position>y<X_position>.tif`

The `X_position` and `Y_position` are integers that determine the position where the pattern was measured on the studied specimen.

The patterns were measured in a triangle raster so the `X_position` and `Y_position` can be iterated as follows:

```
x = i*60 + (j % 2) * 60/2
y = int(j*sqrt(0.75)*60)
```

We have prepared a python script `TEST_RUN/run.py`, that runs the executable on the test data. The only required change to the script is to set the correct path to the executable.

We also prepared a script in `test_data/expand_data.py` that copies some of the test files to create a larger dataset. Just set the size of the dataset directly in the script. In order to run the application on the larger dataset, set the same size in `TEST_RUN/run.py`.

B. Attachments

In the electronic attachment to this thesis we enclose the source codes of our implementation the reference python implementation and a subset of testing data provided by the Department of Physics of Material at Charles University in Prague.