MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Image reconstruction in digital holographic microscopy on GPU

BACHELOR THESIS

Andrej Krejčír

Brno, 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Andrej Krejčír

Advisors:
doc. RNDr. Pavel Matula, Ph.D.
RNDr. Pavel Karas

Acknowledgement

I would like to thank my supervisor doc. RNDr. Pavel Matula, Ph.D. and my consultant RNDr. Pavel Karas for their expert advice and comments during the work on this thesis. Next, I would like to thank everyone who supported me during my work.

Abstract

The aim of the thesis is to implement and optimize chosen image processing algorithms used in digital holographic microscopy on the GPU. The algorithms are 2-D phase unwrapping and polynomial surface fitting. They are described and certain used optimizations are pointed out. The results chapter shows the performance and precision of the GPU implementation compared to CPU on various input images.

Keywords

Digital holographic microscopy, Fourier transform, Phase unwrapping, Least squares method, Polynomial surface fitting, CUDA

Contents

1 Introduction

1.1 Digital holographic microscopy

Digital holographic microscopy is a special form of microscopy used widely in biology to observe living cells and tissues non-intrusively in their environment. The advantage of DHM compared to other types of microscopy is that it produces two images containing two distinct types of information: a light intensity image and a phase shift image. The intensity image shows the intensity of the light that passed through the object and gives information about the transparency of the object. The phase image shows the phase shift between the light beam that passed through the object and the reference beam. The object has a different refractive index than the environment so the light that passes through the former has a different phase than the light that passes through the latter. The phase shift represents optical thickness of the object and can be used to calculate the distance that the light travelled through the object

DHM uses interference patterns from coherent light sources to get information about the object. It works on the principle of holography. A light beam of single frequency is divided into two beams. One is shone through the object and the other travels the same distance through empty space. These two beams interfere together and the interference pattern is recorded by an image sensor. More detailed explanation of DHM can be found in [1].

The image recorded by the microscope must be digitally processed to get the two images of the object. This process should be fast enough to allow real-time observation of the object. In this thesis, these algorithms were implemented on the GPU to get better performance.

Examples of the hologram image and of the intensity and phase image are shown in Figure 1.1 and 1.2, respectively.

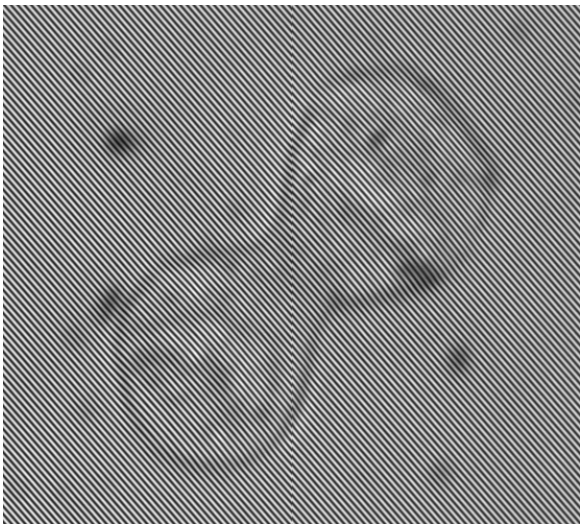


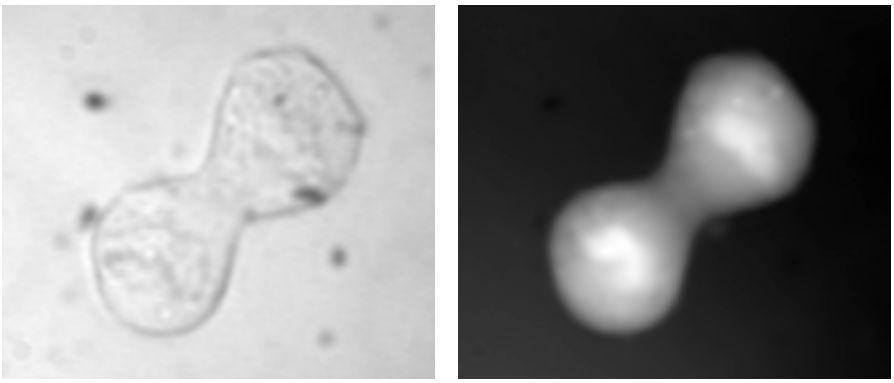
Figure 1.1: Example of a hologram image.

1.2 CUDA and GPU programming model

Modern graphics cards have a programmable processing unit which makes them usable for solving general problems unrelated to real-time graphics or video output. There are several programming languages specialized for writing programs that run on the GPU.

GPU is a massively parallel processor which is able to run thousands of threads. To achieve this, a specific execution model is used, called SIMT — single instruction, multiple threads. This means that one instruction is executed by multiple threads, each operating on different data. The GPU is good for solving data-parallel problems where the task is composed of many instances of the same operation, each on different data. It is not suitable for task-parallel problems, where the tasks may depend on each other using non-trivial synchronization.

The implementation part of this thesis is written in CUDA, but many principles apply to other languages as well, because the programming paradigm is specific for the GPU. CUDA (Compute unified device architecture) is a general purpose computing model and programming language for the GPU. Detailed description of the programming model can be found in [2] and in [3]



(a) Intensity (b) Phase

Figure 1.2: Example of intensity and phase image.

The basic architecture of the CUDA model is the following. The kernel is executed by blocks of threads. Each block contains the same amount of threads. The number can be specified when launching the kernel. Threads are grouped into blocks because only one block can run on one multiprocessor at the same time. This thread hierarchy allows scalability across different graphics cards with different number of multiprocessors. The threads inside one block can exchange data using shared memory, and can be easily synchronized. The global synchronization across blocks is not directly supported due to the fact that it is too expensive for a large number of GPU processors.

The memory hierarchy on the GPU has three layers, each has different access latency. The first one is represented by the global memory. It is accessible by any thread in the kernel and can be as large as the physical memory on the graphics card, but has the biggest latency. The second one is the shared memory. Each block of threads has its own shared memory and no other block can access it. It has much lower latency than the global memory, but its size is limited. The implementation part of this thesis is written for GPUs with compute capability 2.0, where the size of the shared memory can be up to 48 KB. The third memory layer is represented by registers. Each block has a certain amount of memory reserved for registers. This means that the number of registers available to a thread depends on the number of threads in the block. Registers have the lowest latency,

so it is good for performance to keep as much of the data in the registers. However, registers are used only when the access pattern to the data can be determined at compile time. Otherwise, the data is stored in a thread-local part of the global memory, called local memory, which has the same high latency.

The GPU kernels can be optimized by using memory layers with the lowest latency possible, optimal access pattern to memory and optimal size of the thread blocks.

2 Image Processing in Digital Holographic Microscopy

2.1 Fourier transform

Fourier transform is a mathematical transformation of input signal from spacial domain to frequency domain [4]. It is one of the fundamental tools used in digital signal processing. The definition for continuous signal is the following.

$$F(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx$$

The inverse transform is defined as following.

$$f(x) = \int_{-\infty}^{\infty} F(\xi) e^{2\pi \xi x} d\xi$$

Where $f(x)$ is a function $\mathbb{R} \to \mathbb{C}$ in the spacial domain, $x \in \mathbb{R}$ is the spacial coordinate, $F(\xi)$ is a function $\mathbb{R} \to \mathbb{C}$ in the frequency domain and $\xi \in \mathbb{R}$ is the the frequency.

2.1.1 Usage in Digital signal processing

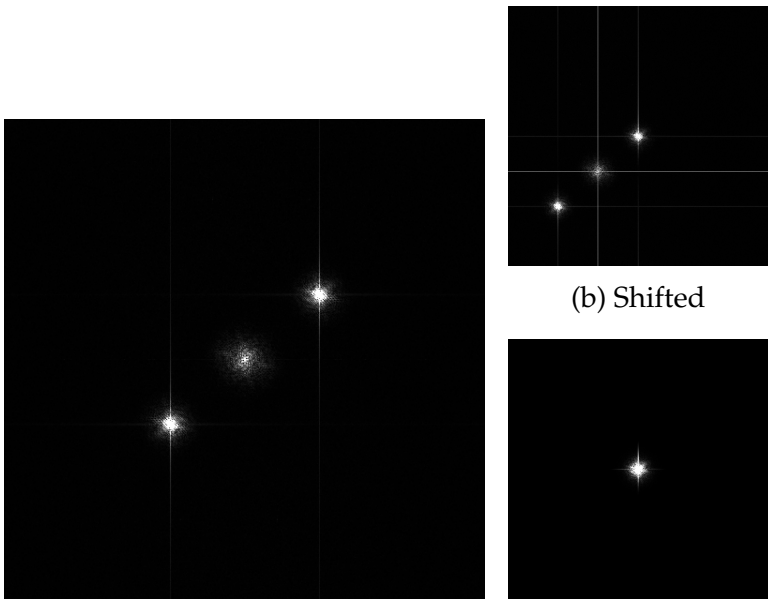
In digital signal processing, the discrete version of Fourier transform is used. The input is a signal sampled at regular frequency and the output is sampled frequency spectrum. In spite of the sampling of both signals, the transform is lossless.

The discrete Fourier transform can be computed efficiently using the method called fast Fourier transform [4]. The method has $\mathcal{O}(NlogN)$ complexity in contrast to the naive method from definition, which has $\mathcal{O}(N^2)$ complexity.

2.1.2 Usage in DHM

Image acquired by the holographic microscope is an interferogram of the incoming light. This image contains one dominant spacial frequency as seen in Figure 1.1. Using Fourier transform, this frequency

is shifted to the center of the spectrum to zero frequency. By this shift, the information modulated in the waves is demodulated. The shifted spectrum is multiplied by a windowing function to remove the other dominant frequencies. The size of the window is as large as possible while still excluding the other peaks in the spectrum. These operations are showed in Figure 2.1.



(a) Original (b) Shifted (c) Windowed

Figure 2.1: Fourier spectrum of the hologram in Figure 1.1

Inverse Fourier transform is applied to the spectrum to obtain the image of the object. It has two components, intensity image and phase image, as seen in Figure 1.2.

There are two artifacts typically present in the phase image. The first one is caused by the fact that the values of the phase lie in the interval $(-\pi, \pi)$. Therefore, any information modulated in the phase image is wrapped around this interval. This introduces sudden steps in phase at the edges of the interval and makes the information non-continuous. The second one is uneven illumination, caused by the nature of encoding of the information in the hologram image.

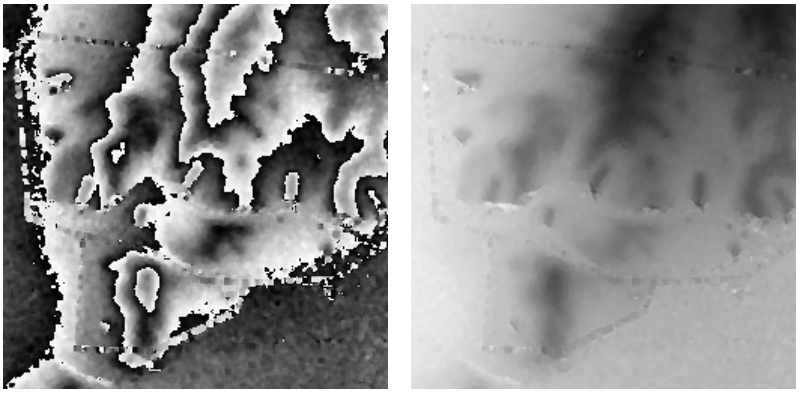
The suppression of the artifacts is essential task in DHM. The following text, describes several methods to achieve this goal.

2.2 Phase unwrapping

There are several applications of digital image-processing techniques which work with images in complex domain. The algorithms often use arctangent function to get the phase of the image, which returns phase wrapped around the interval $(-\pi, \pi)$ and any integer multiples of 2π in the continuous phase signal are lost. The subsequent stages of the image processing pipeline often require a continuous phase image, so an inverse operation to the wrapping has to be applied.

Phase unwrapping techniques are used to reconstruct the original function encoded in the phase. This can be an ill-posed problem if there is noise in the signal or the spacial frequency of the jumps in phase is too high compared to the sampling frequency of the signal. As implied by the Nyquist–Shannon sampling theorem [5], it has to be at least half the image sampling frequency.

In order for these methods to work, an assumption about the original signal is made, requiring that the difference between neighbouring pixels is in the interval $(-\pi, \pi)$. Then, phase unwrapping is computed by integrating wrapped gradients of the phase image. Wrapped and unwrapped phase image can be seen in Figure 2.2.



(a) Wrapped (b) Unwrapped

Figure 2.2: Image from [6]

There are two groups of algorithms solving the phase unwrapping problem:

- Path-following methods, which are based on locally integrating the wrapped phase gradient over the image.
- Minimum-norm methods, which are global integration techniques.

These are described in detail by [7]. One of the path-following methods is implemented in this thesis.

2.2.1 Unwrapping in one dimension

For 1-D signal, phase unwrapping is a simple operation because there is only one possible integration path.

The main idea is to go through all the samples linearly and unwrap them in respect to the previous sample. The process is shown in the following equation.

$$\hat{P}_i = \hat{P}_{i-1} + wrap(P_i - \hat{P}_{i-1}) \tag{2.1}$$

Where P_i is the wrapped signal, $\hat{P}_i$ is the unwrapped signal and function $wrap$ wraps the value around the interval $(-\pi, \pi)$ as follows.

$$wrap(x) = x - 2\pi \left\lfloor \frac{x+\pi}{2\pi} \right\rfloor$$

The first sample of the signal is considered unwrapped $\hat{P}_0 = P_0$. More in-depth explanation of 1-D unwrapping can be found in [8]. Wrapped signal is shown in figure 2.3.

2.2.2 Problems in two dimensions

The technique described in previous subsection can be extended to work for 2-D images. One point is considered as unwrapped and every other point can be unwrapped by integrating wrapped phase gradient on the path from the unwrapped point to the new point. Many possible paths of integration exists in two dimensions, but if the signal is simple and without noise, the result does not depend on the chosen path. [7]

Often, this simple technique does not produce a reasonable result. Due to the noise or a certain topography of the signal, there can

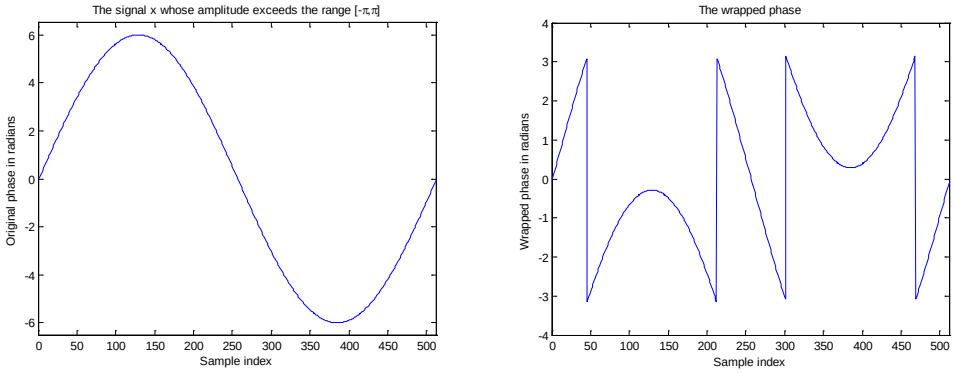


Figure 2.3: Unwrapped and wrapped signal from [8]

be points where the phase is not defined, where both the real and imaginary parts of the complex number are zero. These points can be thought to lie on the crossing of lines with equal wrapped phase as seen on Figure 2.4. Any closed loop in the path of integration that encircles this point has non-zero value. Such paths must not be used during unwrapping because they introduce an error of multiple of 2π to the unwrapped value, and it will propagate through the image, as shown on Figure 2.5. The residue points always appear in pairs. Integral in a loop around one of them has a positive value and integral around the other one has a negative value.

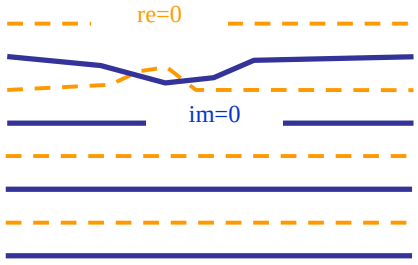


Figure 2.4: Appearance of residue pair from [9]

The exact point in which the phase is not defined may not be present in the sampled image because it can lie on coordinates between the pixels. However, its effect remains.

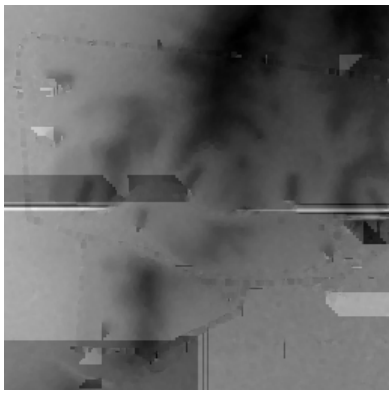


Figure 2.5: Propagation of the error while unwrapping.

There are two main causes why residues appear in the image, as written above. The first one is noise in the image and the second one is the topography of the signal. Residues caused by noise often lie in neighbouring positions and form so-called dipoles. It is relatively easy to detect and pair them. Residues caused by topography are often far away from each other and it is hard to determine exactly how these points should be paired. Such residue pair can be seen on Figure 2.6. Various methods exists to solve this pairing problem, such as pairing the closest points, or pairing based on some a priori knowledge about the original signal.

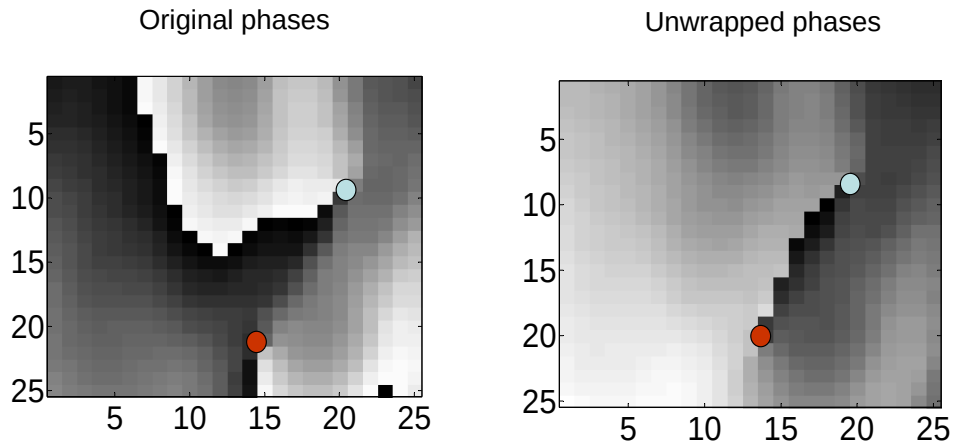


Figure 2.6: Example of residue caused by topography from [9]

2.2.3 Residue detection and pairing

To successfully integrate the phase gradient over the image, the algorithm must avoid paths that would cause inconsistencies, introduce an error and propagate it through the image, as explained in the previous subsection. To achieve this, certain integration paths are avoided based on Goldstein's branch cut algorithm [10], also described in [7]. The algorithm pairs residue points with opposite polarity and prevents any integration path from crossing the line between them.

The first step of the unwrapping algorithm is to detect the position and polarity of residues in the image. The function to test if there is a residue point at certain coordinates is shown in Algorithm 2.1. To detect the position of the point, the gradient is integrated in the smallest loops of four pixels. If the value of the integral is nonzero a residue point is found and its polarity is equal to the sign of the value of the integral.

Algorithm 2.1 Residue detection at coordinates (x, y)

function DETECTRESIDUE(x,y)
  $V_0 := img(x, y)$
  $V_1 := img(x + 1, y)$
  $V_2 := img(x + 1, y + 1)$
  $V_3 := img(x, y + 1)$
  $D := wrap(V_1 - V_0) + wrap(V_2 - V_1)$
  $D := D + wrap(V_3 - V_2) + wrap(V_0 - V_3)$
  if $D \geq 2\pi$ **then**
    Positive residue found at (x, y)
    return $POSITIVE\_POINT$
  end if
  if $D \leq -2\pi$ **then**
    Negative residue found at (x, y)
    return $NEGATIVE\_POINT$
  end if
  return NO_POINT
end function

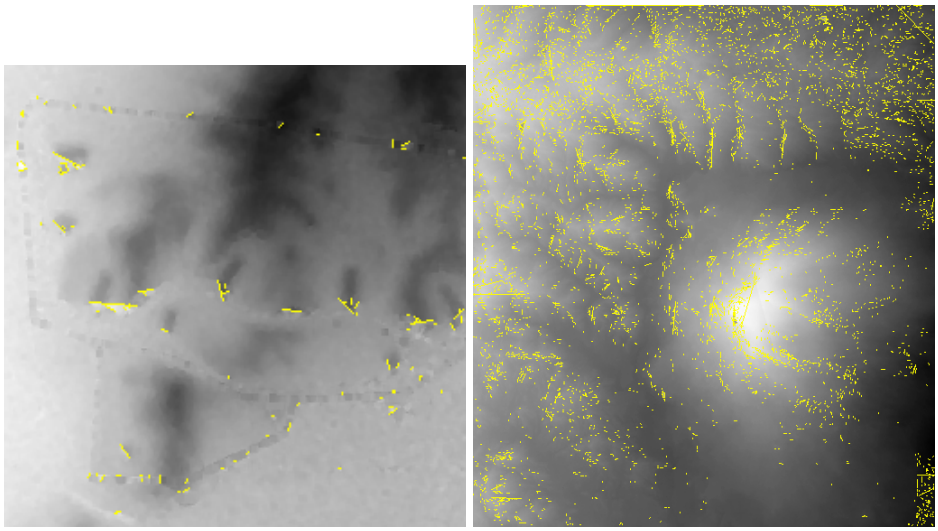
Where the $wrap()$ function wraps the value to the interval $(-\pi, \pi)$.

After the points are found, they need to be paired. It is not clear

which positive points should be paired with which negative points. Various pairing methods can be chosen. The algorithm implemented in this thesis tries to pair residues of opposite polarity with the smallest distance from each other.

It is easy to pair residues of opposite polarity next to each other or very close to each other. They are considered dipoles caused by noise, and because they are close, there is small possibility of error due to wrong pairing. The other points are paired based on minimal distance from each other. The optimal solution may not always be found, but the results are reasonable enough. Certain maximum pairing distance is defined and only points up to this distance are paired. The rest of the points are considered to have the corresponding pairing point outside the image, so they are paired with the nearest edge of the image.

The pairs of residues define lines, called brunch cuts, which must not be crossed by an integration path. These are shown in Figure 2.7.



(a) Unwrapped, Figure2.2 (b) Unwrapped, Figure4.5

Figure 2.7: Branch cuts are shown in yellow.

2.2.4 Integration by flood-fill

After pairs of residues have been found, it is safe to integrate the wrapped gradient of the image. The result will be independent on the choice of integration path, as long as it does not cross any branch cut.

The integration is based on the idea of the flood-fill algorithm. Which gradually finds paths from one specific point to all the other points in the image, avoiding branch cuts and unwrapping pixels on the way.

One point is chosen as the starting point and it is marked as unwrapped. Subsequently, all pixels in its 4-neighbourhood [4] that don't lie on a branch cut are unwrapped using the equation 2.1

2.3 Uneven background removal

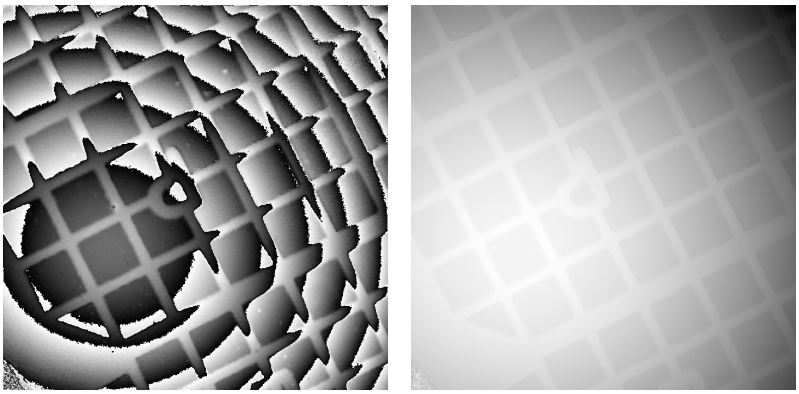
The unwrapped phase image which was acquired by the methods described in previous sections is corrupted by an uneven illumination. The term "illumination" may not be accurate, because the effect is not caused by different light intensity. It is caused by the way the phase image is encoded in the interferogram.

This effect can be seen in Figure 2.8, where it is clear that it is not just a linear gradient. This uneven illumination should be removed to get a quantifiable phase image of the object.

The background can be sufficiently approximated by a third-order two-dimensional polynomial function. The approximation is abstract enough so that it fits only the illumination of the image, not the details of the object. For better fit, only the pixels corresponding to the background can be used. After the polynomial is computed, it is subtracted from the image, so that the background corresponds to zero value in the final image. The result can be seen in Figure 2.9.

2.3.1 Used numerical methods

The coefficients of the polynomial surface are chosen to minimize selected error function. The result is that the surface approximates the image as closely as possible with a given number of coefficients.



(a) Wrapped (b) Unwrapped

Figure 2.8: Phase of a test image.

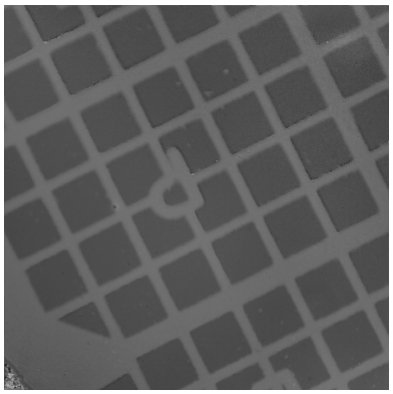


Figure 2.9: Background removed from the image in Figure 2.8.

The polynomial function has the following form.

$$P_C(x,y) = C^T V_{x,y}, \text{ where } C = \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \\ C_7 \\ C_8 \\ C_9 \end{pmatrix}, V_{x,y} = \begin{pmatrix} 1 \\ x \\ y \\ x^2 \\ xy \\ y^2 \\ x^3 \\ x^2y \\ xy^2 \\ y^3 \end{pmatrix}$$

The chosen error function in this case is the sum of square differences of the pixel value and the value of the polynomial surface at each point.

$$E(C) = \sum_{x,y} (P_C(x,y) - I_{x,y})^2$$

Where $I_{x,y}$ is the value of pixel at coordinates (x, y).

The polynomial surface is a linear function of its coefficients, so finding the optimal coefficients is the Linear Least Squares problem [11]. The optimal coefficients lie in the global minimum of the error function.

Gradient of the error function is following.

$$\begin{aligned}
\nabla E(C) &= \sum_{x,y} \nabla \left[(C^T V_{x,y} - I_{x,y})^2 \right] \\
\nabla E(C) &= \sum_{x,y} 2(C^T V_{x,y} - I_{x,y}) \cdot \nabla (C^T V_{x,y} - I_{x,y}) \\
\nabla E(C) &= 2 \sum_{x,y} (C^T V_{x,y} - I_{x,y}) V_{x,y}
\end{aligned}$$

By setting the gradient to zero, we get the coordinates of local minima.

$$\begin{aligned}
\nabla E(C) &= 0 \\
\sum_{x,y} (C^T V_{x,y} - I_{x,y}) V_{x,y} &= 0 \\
\sum_{x,y} V_{x,y} C^T V_{x,y} - V_{x,y} I_{x,y} &= 0 \\
\sum_{x,y} V_{x,y} \left(V_{x,y}^T C \right)^T &= \sum_{x,y} V_{x,y} I_{x,y} \\
\left(\sum_{x,y} V_{x,y} V_{x,y}^T \right) C &= \sum_{x,y} V_{x,y} I_{x,y} \\
\mathbf{M} C &= Y
\end{aligned}$$

Where $V_{x,y} V_{x,y}^T$ is the outer product of column vectors $V_{x,y}$. The resulting equation is a linear system of equations, where $\mathbf{M}$ is a square

symmetric and positive-definite matrix. The system has exactly one solution, because **M** has a positive determinant.

The matrix **M** and the vector Y are calculated as a sum through all the used points in the image. Because not all points in the image are used, the Matrix **M** cannot be expressed analytically.

The solution is the vector C of optimal coefficients. To solve the system, matrix **M** is decomposed into the product of two triangular matrices using Cholesky decomposition [12]. Then the system is solved for each matrix in sequence. Solving a linear system with a triangle matrix is trivial. So it is preferable to use this solution, in contrast to another, for example the Gaussian elimination, which is computationally more expensive, and more difficult to implement on GPU.

Cholesky decomposition decomposes a matrix to the product of one triangle matrix and its transpose.

$$\mathbf{M} = \mathbf{L}\mathbf{L}^T$$

Not every matrix can be decomposed this way. The matrix must be symmetric and positive-definite. These conditions apply to the matrix **M**, so this method is used.

The decomposition is defined as follows:

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{00} & L_{10} & L_{20} \\ 0 & L_{11} & L_{21} \\ 0 & 0 & L_{22} \end{pmatrix}$$

From this, the equations for $L_{i,j}$ are following.

$$L_{i,i} = \sqrt{A_{i,i} - \sum_{k=0}^{i-1} L_{i,k}^2}$$

$$L_{i,j} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=0}^{j-1} L_{i,k} L_{j,k} \right)$$

After decomposing the matrix, the equation $\mathbf{M}C = Y$ can be solved in two steps as following.

$$\mathbf{L}X = Y$$

$$\mathbf{L}^T C = X$$

The first equation is in the form:

$$\begin{pmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \end{pmatrix}$$

The solution for elements of vector X is the following.

$$X_i = \frac{1}{L_{i,i}} \left(Y_i - \sum_{j=0}^{i-1} X_j L_{i,j} \right)$$

Similar solution can be found for the second equation, only the loop with index i goes from bottom up.

Using this method, the optimal coefficients are computed.

3 Implementation of Described methods

3.1 Existing implementations of Fourier transform on GPU

The fast Fourier transform was implemented and optimized for GPU in the library CUFFT, which is a part of the CUDA Toolkit and is usually installed with CUDA drivers. Its API is modelled after FFTW [13], one of the most efficient and widely used CPU implementations. More information can be found in [14].

3.2 Phase unwrapping algorithm

The algorithm implemented as a part of this thesis is a parallel version of branch-cut method, based on principles of Goldstein's algorithm [10] also described in [7].

The whole algorithm is split into the following parts. Detection of residue points, pairing the residues, rasterization of pairs, integration of phase gradient using flood-filling and interpolation of branch cuts. Each part consists of its own kernel because they operate mostly on different type of data and could not be merged into a single kernel. For this reason, only pairing the points which are few pixels away from each other, called dipoles, is done in the same kernel as detecting residues. The reason is explained in the following subsection.

For storing pairs and points in memory an array is used. It is implemented as a buffer of a sufficient size and one 32-bit unsigned integer pointing past the last element. In the worst case, the number of residue points cannot be higher than the number of all points in the image, so a sufficient buffer size for the array is the size of the image. When a thread wants to add elements to the back, it allocates the needed number of elements by increasing the back pointer using atomicAdd instruction. This ensures deterministic behaviour if multiple threads want to add elements to the array at the same time.

To measure the distance between the residue points, the supre-

mum norm is used.

$$||P - N||_\infty = \max(|P_x - N_x|, |P_y - N_y|)$$

Where P is the positive residue point and N is the negative point. The value of the distance is an integer, because the coordinates of the points are only integers. The fact is used in the pairing algorithm.

All kernels use templates wherever possible. Because they provide more information to the compiler to optimize the code, like to unroll the cycles, reorder instructions and inline function calls.

3.2.1 Finding the residue points

The idea how residue points are found in the image is described in previous chapter. The implementation of the idea is straightforward. One thread is run for each pixel and executes Algorithm2.1. If a residue point is detected, the coordinates are stored to appropriate array in the global memory.

Various optimizations of this kernel have been tested. For example, loading a block of image into the shared memory and finding the residues there, which reduces access to the global memory to approximately $1/4$. However, using the shared memory only to detect residues brings no significant performance boost. This may be caused by the atomicAdd operation used in each thread to allocate space in the global array.

The implemented kernel finds the residues using the method described above and pairs points that are close to each other. It takes advantage of the information about the surrounding pixels, which is lost once the points are stored into an array in undefined order. Only residues in the same block are paired in this stage. Still, the global performance gain is significant.

The kernel works as shown in Algorithm 3.1. No atomic operation is needed because the surrounding pixels are checked in the same order for every point and threads are synchronized after the check. The condition $tx, ty \in \{dist - 1, ..., BlockSize - dist\}$ ensures that, the distance from the positive point to every edge is at least $dist - 1$ so no negative point from other block is closer than the found negative point from this block, otherwise it would cause incorrect pairing. The mask in shared memory has one pixel border with no

negative points so no additional boundary checking is needed. The $maxDipoleDistance$ was experimentally set to be 3 pixels in this thesis.

Algorithm 3.1 Residue detection and dipole pairing kernel

function RESIDUEDETECTIONKERNEL
  SHARED $mask[BlockSize + 2][BlockSize + 2]$
  $mask(tx, ty) :=$ DetectResidue($imgx$, $imgy$)
  syncThreads()
  if $mask(tx, ty) == POSITIVE\_POINT$ **then**
    for $dist := 1$ do $maxDipoleDistance$ **do**
      if $tx, ty \in \{dist - 1, ..., BlockSize - dist\}$ **then**
        for all $(x, y) : ||(x, y) - (tx, tx)|| == dist$ **do**
          if $mask(x, y) == NEGATIVE\_POINT$ **then**
            $mask(x, y) := NO\_POINT$
            $mask(tx, ty) := NO\_POINT$
            Found residue pair: (tx, ty) , (x, y)
          end if
          syncThreads()
        end for
      end if
    end for
  end if
end function

Where tx, ty are coordinates of the thread inside the block, $imgx, imgy$ are coordinates of the thread in the image, $BlockSize$ is one dimension of the square block and $||X||$ is the supremum norm.

The kernel with pairing of dipoles takes more time, but greatly reduces the overall time of the whole unwrapping algorithm. Because dipoles are caused by noise, noisy areas in the image contain a lot of residues close together. Pairing them in the pairing stage would require much more time because the performance of the pairing kernel is proportional to the number of residue points found. On the other hand, the performance of this kernel depends only on the size of the image.

3.2.2 Pairing the residue points

Pairing the residues is split into two parts, pairing dipoles, and pairing the rest of the points. This subsection describes the pairing of points, which weren't paired in the previous part.

The pairing kernel is called iteratively. Each iteration pairs points at specified distance away from each other. The distance is increased in each iteration and the loop stops when a maximum distance is reached, which can be set empirically. Lower values have better performance and higher values may produce more optimal result in some cases. In this thesis the value is set to $\max(width, height)/4$.

A thread is run for each positive point and it linearly scans all the negative points, that haven't been paired in previous iterations. The information if a point is free to be paired or was already used is stored in a mask in global memory. When a negative point is found at a specified distance from the positive, its mask is changed to "used" by the atomic operation atomicExch. This avoids a race condition when more threads want to create a pair with the same negative residue, only one of them succeeds. The others continue checking other points.

The algorithm as explained here is not very efficient. There are two things that can be successfully optimized without changing the logic of the algorithm.

First let us consider what influences the performance of the kernel. A thread is run for each positive point and right after it starts, it checks if the point is free. If not, the thread ends. However, it makes performance better only if all 32 threads in the warp evaluate this condition as false finishing the whole warp at once. Otherwise, all 32 threads are active even if just one of them processes a free positive point. So in the worst case, the number of free points can be 32 times less than the number of all points and the performance will be the same as if all were being processed.

Secondly, the performance also depends on the number of negative points. There is no divergence because all threads in the warp access the same negative point. If the negative point is not free it is simply skipped, but the check still requires a memory operation.

To gain performance, the arrays can be reorganized and shrunk between the iterations of the kernel so that they only contain the

unpaired points. This reduces the number of thread blocks needed by the kernel and also each warp is better utilized, because the free points are close together.

Two parameters of this shrinking were tested and optimal values were found experimentally. The first is the number of iterations of the pairing kernel between the shrink kernels, as it is not optimal to shrink after each pairing iteration. A good value was found to be 3. The second parameter is the smallest array size for which the shrink kernel is run, because it has little effect to shrink small arrays. The value of 32 is the minimum for this parameter, because the whole array is processed by one warp and no performance is gained by shrinking it. Experimentally, a good value was found to be 64.

Another optimization is to minimize the number of iterations of the pairing kernel. This means to minimize the number of different distances for which the kernel is called. Iterating through all possible distances from 1 to the maximum pairing distance is inefficient, but if a distance was skipped and some two points were the given distance away from each other they would not be paired. So an additional parameter added to the kernel is a pointer to an integer in the global memory, storing the distance that will be used in the next iteration. Before the kernel is run, the number is set to the maximum distance. When a thread finds a point whose distance is less than the stored next distance and more than the current pairing distance, it updates the next distance using atomicMin instruction. After the kernel has finished, the number is copied to the host memory and used in the next iteration. This can greatly increase the performance, because it skips unused distances.

3.2.3 Rasterization of pairs

The rasterization kernel uses one thread per line and each line is rasterized using the Bresenham line algorithm[15], example is shown in Figure 3.1. The kernel is highly divergent, because each thread rasterizes different line and writes to a different location in the global memory in a loop with different length. However, its performance is acceptable for average input image.

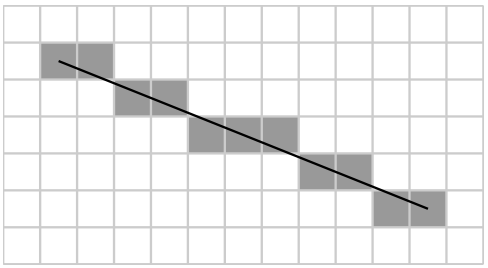


Figure 3.1: Rasterized line using Bresenham algorithm

3.2.4 Flood-fill gradient integration

The integration of wrapped phase gradient is implemented on the basis of flood-fill method. The basic implementation idea is to process the whole image iteratively. In each iteration more pixels are unwrapped. Each pixel can be in one of the following four states.

- Unprocessed pixels, they contain the wrapped phase and do no lie on a branch cut.
- Branch cut pixels, they were marked by the pair rasterization kernel and are ignored by this kernel.
- Active pixels, they are in an intermediate state. Their phase value is unwrapped, but some of the surrounding pixels may be in the unprocessed state. These pixels form an edge between processed and unprocessed pixels.
- Processed pixels, their value is unwrapped and they are not surrounded by any unprocessed pixels.

The state of each pixel is stored in a mask in the global memory. An example of this mask is showed in Figure 3.2.

Before the first iteration, one unprocessed pixel is marked as active, for example, the one in the center of the image. Then the kernel is called iteratively and each thread runs Algorithm 3.2.

No atomic operation is needed because if more threads unwrap the same pixel from different neighbours concurrently, they compute the same value regardless of the order of reads and writes to the memory location of the pixel. This is caused by two facts. Firstly, the

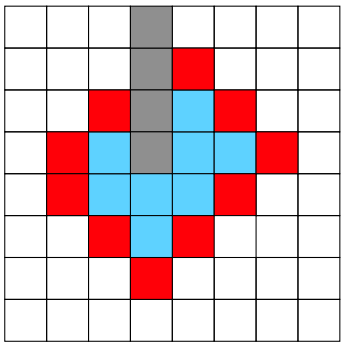


Figure 3.2: Pixel states: white - unprocessed, grey - branch cut, red - active, blue - processed

result of the gradient integral is independent on the integration path taken to the pixel. Secondly, the unwrapping operation called on an already unwrapped pixel does not change its value.

The kernel is called in a loop until the $repeatOut$ value is false. When the loop ends the image is fully unwrapped or the areas of unprocessed pixels are enclosed by branch cut pixels. In the worst case, the number of iterations of this trivial kernel is proportional to the number of pixels in the image.

A more optimal kernel to solve this problem can be used, taking advantage of the low latency of shared memory. The image is divided into regular blocks and each block is processed by one thread block. So the algorithm has two layers, block layer and pixel layer, as shown in Figure 3.3. A pseudo-code is shown in Algorithm 3.3.

On the block layer, each block can be in one of two states, active and not active. After a block is processed its state is set to not active and if a certain condition is met, its four neighbouring blocks are activated. The condition will be described later. The kernel is called iteratively and in each iteration active blocks are processed and their neighbours are activated.

The same block can be activated multiple times, because there might be multiple areas of pixels separated by branch cuts inside one block and these areas can be connected through multiple other blocks so to unwrap the block successfully, it has to be activated more than once. This can cause problems because all blocks run in parallel and there can be a case where two blocks next to each other are active

Algorithm 3.2 Basic flood-fill kernel

function FLOODFILLKERNEL($repeatOut$)
  $mask := maskImg(imgx, imgy)$
  if $mask == ACTIVE$ **then**
    $val := phaseImg(imgx, imgy)$
    ProcessPixel($imgx + 1$, $imgy$, val)
    ProcessPixel($imgx - 1$, $imgy$, val)
    ProcessPixel($imgx$, $imgy + 1$, val)
    ProcessPixel($imgx$, $imgy - 1$, val)
    $maskImg(imgx, imgy) := PROCESSED$
    $repeatOut := TRUE$
  end if
end function
function PROCESSPIXEL(x, y, $prevValue$)
  if $maskImg(x, y) == UNPROCESSED$ **then**
    $val := phaseImg(x, y)$
    $newVal := prevValue + wrap(val - prevValue)$
    $phaseImg(x, y) := newVal$
    $maskImg(x, y) := ACTIVE$
  end if
end function

Where $imgx, imgy$ are the image coordinates of the thread and the $wrap()$ function wraps the value to the interval $(-\pi, \pi)$. No boundary checking is shown in this algorithm.

at the same time and one of them wants to activate the other. There can be a case when the block is activated only once, not twice. To solve this problem, the mask which stores the state of the block is be double-buffered, one input and one output mask. After each kernel iteration, they are swapped and the new output mask is cleared.

On the pixel layer, a similar unwrapping algorithm is used as the one in previous kernel, but run on a small block in the shared memory.

In the kernel, the function UnwrapFromNeighbours() unwraps and activates pixels on the border, which are next to a processed pixel in the other block. The function ActivateNeighbours() activates neighbouring block if at least one unprocessed pixel in the block can be unwrapped from this block.

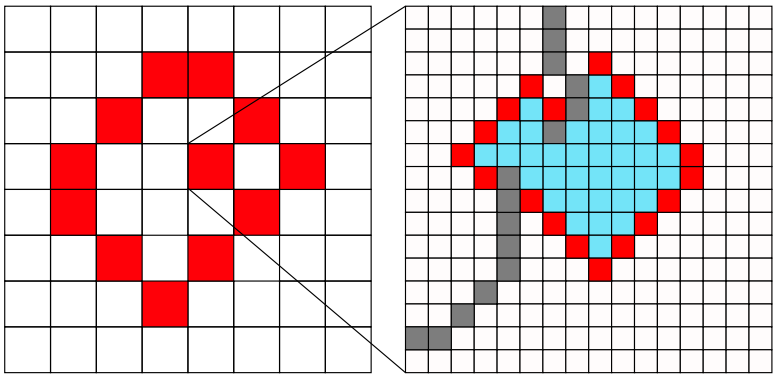


Figure 3.3: Block layer and pixel layer

This optimization makes the performance about three times better. Different sizes of the block have been tested, the size of 16x16 is fastest on the tested GPU.

3.2.5 Interpolation of branch cuts

The interpolation is quite straightforward. For each pixel lying on the branch cut, the weighted average value of the unwrapped neighbours is computed and the result is written as value of the pixel. The weights are shown in the following matrix.

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 0 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

3.3 Least squares method on GPU

The algorithm has the following parts.

- Computation of the needed coefficients to construct the linear system of equations as described in the previous chapter.
- Solving the system and computing the coefficients of the polynomial.
- Subtracting the polynomial surface from the image to remove background.

Algorithm 3.3 Block flood-fill kernel

function FLOODFILLBLOCKKERNEL($repeatOut$)
  SHARED $sPhaseImg[BlockSize][BlockSize]$
  SHARED $sMaskImg[BlockSize][BlockSize]$
  if $blockMask(bx, by) == ACTIVE$ **then**
    Load block from $phaseImg$ to $sPhaseImg$
    Load block from $maskImg$ to $sMaskImg$
    UnwrapFromNeighbours($sPhaseImg$, $sMask$)
    SHARED $sRepeat := FALSE$
    repeat
      FloodFill($sPhaseImg$, $sMaskImg$, $sRepeat$)
      syncThreads()
    until $sRepeat == FALSE$
    Store $sPhaseImg$ block to $phaseImg$
    Store $sMaskImg$ block to $maskImg$
    ActivateNeighbours($sMask$, $repeatOut$)
  end if
end function

Where bx, by are coordinates of the block in the grid, $phaseImg$ is the phase image in global memory, $maskImg$ is the mask image in the global memory, function FloodFill() is similar to the kernel in Algorithm 3.2 and functions UnwrapFromNeighbours() and ActivateNeighbours() are described in Algorithm 3.4.

3.3.1 Reduction of the coefficients

As explained in previous chapter, to find the coefficients of the polynomial surface, a system of linear equations has to be solved. This system is in the form $\mathbf{M}C = Y$ where the values of matrix $\mathbf{M}$ and vector Y depend on all used pixels in the image.

The basic idea is to compute the matrix and the vector for each pixel in the image and then sum them together into a single matrix and a single vector. This sum is implemented as parallel reduction.

The reduction on CUDA is implemented in the following way. In the first pass, each thread computes its value and keeps it in a register. Each block of threads performs the reduction independently using the shared memory and the single result is saved to global memory. In the subsequent passes, the values are read, the reduc-

Algorithm 3.4 Functions used in Algorithm 3.3

function UNWRAPFROMNEIGHBOURS($sPhase$, $sMask$)
  if (tx, ty) is border pixel of the block **then**
    (ox, oy) is the pixel on the other side of the block edge
    if $sMask(tx, ty) == UNPROCESSED$ **and**
     $maskImg(ox, oy) == PROCESSED$ **then**
      $val := sPhase(tx, ty)$
      $nearVal := phaseImg(ox, oy)$
      $sPhase(tx, ty) := nearVal + wrap(val - nearVal)$
      $sMask(tx, ty) := ACTIVE$
    end if
  end if
end function
function ACIVATENEIGHBOURS
  if (x, y) is border pixel of the block **then**
    (ox, oy) is the pixel on the other side of the block edge
    if $maskImg(x, y) == PROCESSED$ **and**
     $maskImg(ox, oy) == UNPROCESSED$ **then**
      $blockMask(otherBlockX, otherBlockY) := ACTIVE$
      $repeatOut := TRUE$
    end if
  end if
end function

tion is performed in the shared memory, and the results are stored in the global memory. The step is repeated until only a singe value remains. Detailed description can be found in [16].

The effective reduction in shared memory is shown in Listing 3.1 where `N` is the number of values in shared memory that are being reduced, `tx` is the thread number, `val` is the thread's value stored in register and `sharedVal` is the array in shared memory.

To represent the matrix **M**, only 28 values are needed, the rest of the 100 are duplicates of some value from the 28. This reduces the amount of needed memory for each thread and speeds up the reduction. The number 28 comes from the observation, that each value in the matrix is the product of two values form the vector $V_{x,y}$. The largest sum of exponents of x and y in the vector is 3, so the largest sum of exponents in the matrix **M** is 6. The number of combinations

Listing 3.1: Parallel reduction in shared memory

```
template<int N>
__device__ inline void blockReduction
    (uint32_t tx, Val& val, Val* sharedVal)
{
    if((tx >= N / 2) && (tx < N))
        { sharedVal[tx - N/2] = val; }
    if(N > 32) { __syncthreads(); }
    if(tx < N / 2)
        { val += sharedVal[tx]; }
    blockReduction<N/2>(tx,val,sharedVal);
}

template<>
__device__ inline void blockReduction<1>
    (uint32_t, Val&, Val*) {}
```

of exponents with sum up to 6 is: $(6+1)(6+2)/2 = 28$. The matrix coefficients for each pixel are in the following form.

$$\left(1, x, y, x^2, xy, y^2, ..., x^6, x^5y, x^4y^2, x^3y^3, x^2y^4, xy^5, y^6\right)$$

Another 10 numbers are needed to represent the vector Y in the system. They have the following form.

$$Y = I_{x,y}\left(1, x, y, x^2, xy, y^2, x^3, x^2y, xy^2, y^3\right)$$

where x, y are coordinates of the pixel and $I_{x,y}$ is the value of the pixel at those coordinates.

In total, a vector of 38 values is computed for each pixel and subsequently these vectors are summed together.

3.3.2 Cholesky decomposition on GPU

Implementation on GPU has lower performance than the same implementation running on CPU, but it is included for completeness. The reason is that the matrix is small and Cholesky decomposition is mostly sequential algorithm. However, this part takes very little time compared to other, more time-consuming parts so there is no reason to optimize it. The parallel version uses only ten threads, one

Listing 3.2: Matrix decomposition on GPU

```
val_t accum = 0.0f;
for(long j = 0; j < 10; ++j)
{
    if(tid == j)
        { sMat[j][j] = sqrt(sMat[j][j] - accum); }
    __syncthreads();
    if(tid > j)
    {
        val_t diagVal = sMat[j][j];
        val_t accum2 = 0;
        for(long k = 0; k < j; ++k)
            { accum2 += sMat[tid][k]*sMat[j][k]; }

        val_t val = (sMat[tid][j] - accum2)/diagVal;
        sMat[tid][j] =  val;
        accum += val * val;
    }
        __syncthreads();
}
```

for each row of the matrix. The implementation is in the Listing 3.2. Columns are processed in parallel and threads are synchronized between columns. They accumulate needed values form the previous columns. The synchronization is implicit, because all threads are in one warp.

After the matrix is decomposed into multiplication of two lower triangle matrices, the system is solved first for the L matrix and then for L^T. The implementation runs one thread for each row and synchronizes between columns, similarly to the decomposition.

3.3.3 Background removal

After computing coefficients of the polynomial surface approximating background, the value of the surface is subtracted from the image at each pixel. The implementation is straightforward and the kernel is as simple as possible to enable the compiler to do optimizations.

4 Results and performance analysis

The performance of each GPU algorithm is compared to the performance of a CPU algorithm. The CPU algorithms are mostly equivalent sequential versions of the GPU kernels. They are single-threaded and no vector instructions were used.

All tests were run on a system with Intel Core 2 Quad CPU Q6600 2.40GHz and GeForce GTX 470.

4.1 Performance of CUFFT compared to CPU

The performance of CUFFT compared to MKL FFT implementation [17] is shown in the Figure 4.1.

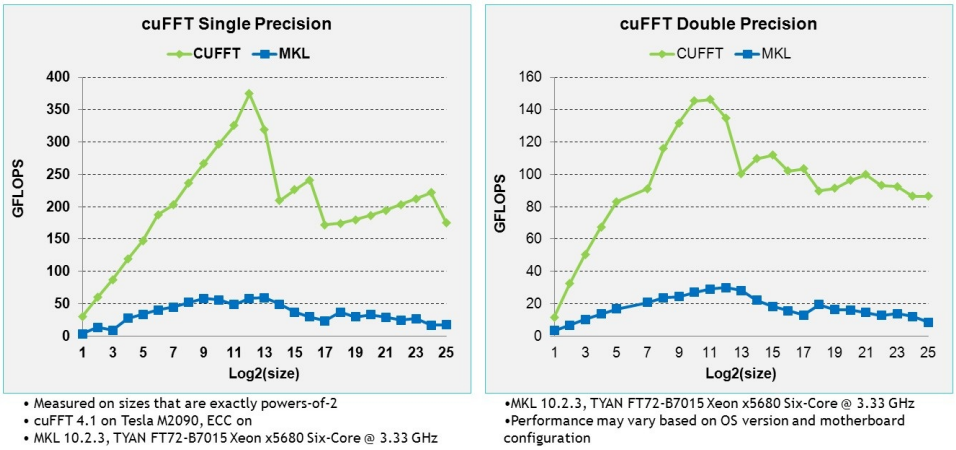


Figure 4.1: Performance of CUFFT compared to MKL, picture from [18]

4.2 Phase unwrapping on GPU and CPU

The performance of GPU and CPU phase unwrapping algorithm is shown in the following tables. More input images were created from one image, by scaling the values by an integer factor and wrapping them. This creates images with higher gradients and more phase

jumps. When double precision numbers are used, the performance is nearly unaffected, except for the memory transfer of the image, which takes twice as long.

Each table contains the time of individual stages of the algorithm. The difference in performance between the single and double precision algorithms on the CPU is negligible, so only the double precision is shown. Table 4.1 shows the performance on a typical image acquired by the microscope. Table 4.2 and 4.3 show the performance on an image with higher resolution. Table 4.4 shows the performance on an image with large number of residue points. For images with many residue points, the pairing kernel takes considerable part of the total time. The second time-consuming part is the flood-fill integration, whose performance depends on the size of the image.

Table 4.5 shows the precision of the GPU implementation compared to the reference CPU double precision implementation. Results are compared using mean square error and peak signal to noise ratio. The maximum signal value for computing PSNR is chosen to 255. Higher values of the error in the fourth row are caused by different pairing of the residues on the GPU compared to CPU.

Table 4.1: Performance of phase unwrapping the image in Figure 4.4 with scaling factor 15.

With the resolution 398 x 299 and 55 residue pairs.

	GPU Float	GPU Double	CPU Double
Memcpy to GPU	0.22 ms	0.40 ms	-
Residue detection	0.21 ms	0.20 ms	11.13 ms
Residue pairing	0.95 ms	0.94 ms	0.06 ms
Rasterization	0.02 ms	0.02 ms	$<$ 0.01 ms
Flood-fill	1.90 ms	1.83 ms	6.18 ms
Interpolation	0.02 ms	0.02 ms	0.24 ms
Memcpy from GPU	0.27 ms	0.51 ms	-
Total time	4.41 ms	4.99 ms	18.09 ms

Table 4.2: Performance of phase unwrapping the image in Figure 4.2

With the resolution 702 x 701 and 130 residue pairs.

	GPU Float	GPU Double	CPU Double
Memcpy to GPU	0.77 ms	1.48 ms	-
Residue detection	0.78 ms	0.74 ms	44.87 ms
Residue pairing	0.83 ms	0.82 ms	0.10 ms
Rasterization	0.02 ms	0.02 ms	0.01 ms
Flood-fill	5.39 ms	5.19 ms	28.13 ms
Interpolation	0.05 ms	0.05 ms	0.98 ms
Memcpy from GPU	1.06 ms	2.08 ms	-
Total time	11.23 ms	13.39 ms	75.56 ms

Table 4.3: Performance of phase unwrapping the image in Figure 4.3

With the resolution 850 x 640 and 1 residue pair.

	GPU Float	GPU Double	CPU Double
Memcpy to GPU	0.84 ms	1.63 ms	-
Residue detection	0.86 ms	0.81 ms	48.81 ms
Residue pairing	0.11 ms	0.11 ms	$<$ 0.01 ms
Rasterization	0.01 ms	0.01 ms	$<$ 0.01 ms
Flood-fill	5.83 ms	5.62 ms	28.71 ms
Interpolation	0.05 ms	0.05 ms	1.06 ms
Memcpy from GPU	1.17 ms	2.29 ms	-
Total time	11.18 ms	13.91 ms	79.40 ms

4.3 Least squares method

In this section, we show the performance of the polynomial fitting algorithm and the background removal. The performance of the computation of coefficients is better on CPU because the Cholesky decomposition is mostly sequential algorithm and the matrix, which is decomposed is relatively small, to the GPU is not fully utilized. The speed of the algorithm depends only on the size of the image. The tests show that the reduction on GPU is much slower when using double precision numbers. The reason is that the reduction kernel stores the 38 intermediate coefficients in registers, but if double precision is used, not all coefficients fit into registers and some of them

Table 4.4: Performance of phase unwrapping the image in Figure 4.5

With the resolution 512 x 578 and 6050 residue pairs.

	GPU Float	GPU Double	CPU Double
Memcpy to GPU	0.49 ms	0.92 ms	-
Residue detection	0.52 ms	0.50 ms	33.17 ms
Residue pairing	3.65 ms	3.63 ms	52.79 ms
Rasterization	0.06 ms	0.06 ms	0.25 ms
Flood-fill	4.15 ms	3.95 ms	15.81 ms
Interpolation	0.09 ms	0.11 ms	1.29 ms
Memcpy from GPU	0.65 ms	1.27 ms	-
Total time	11.01 ms	12.65 ms	105.26 ms

Table 4.5: Precision of the GPU phase unwrapping method, compared to the reference CPU implementation

	Float		Double	
	MSE	PSNR	MSE	PSNR
Figure 2.8	0.51	51.06	0.51	51.05
Figure 4.3	10^{-5}	97.00	10^{-5}	97.00
Figure 4.4	10^{-14}	186.32	10^{-32}	358.82
Figure 4.5	2.21	44.68	1.84	45.48

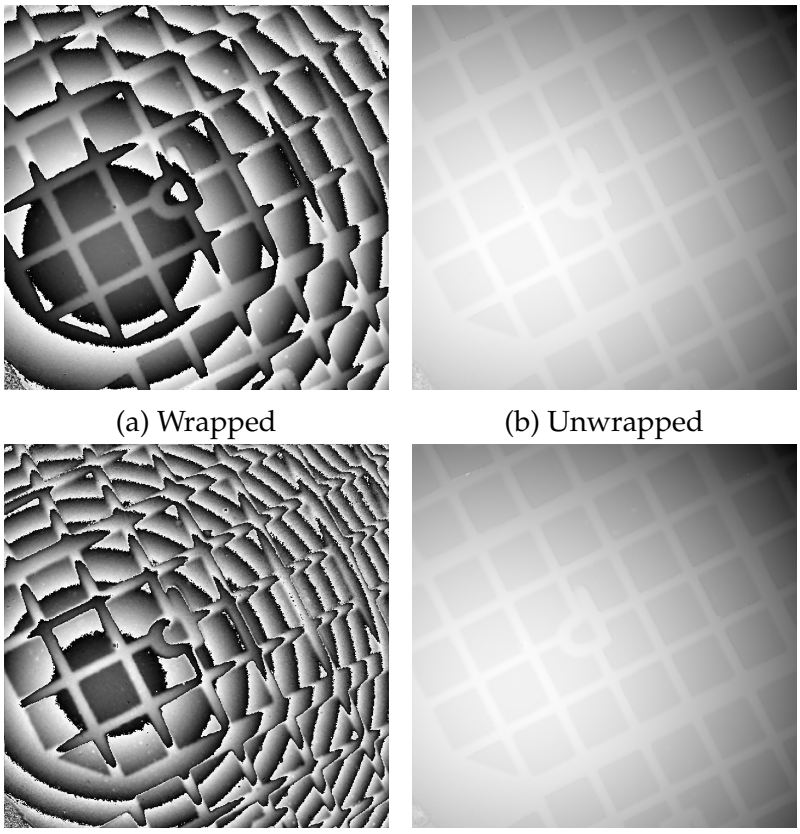
are stored in local memory, which introduces considerable latency.

Table 4.6 and Table 4.7 show the performance on a bigger picture. Table 4.8 shows the performance on a typical image resolution acquired from the microscope. The resulting images are shown in Figure 4.6.

The precision of the GPU algorithm is compared to the reference implementation the same way as in previous section. Table 4.9 shows the results.

4.4 Whole pipeline

The advantage of running the whole pipeline on GPU is that very little memory transfer from host to device and the other way is needed during computation. The transfer can take milliseconds as shown



(a) Wrapped (b) Unwrapped

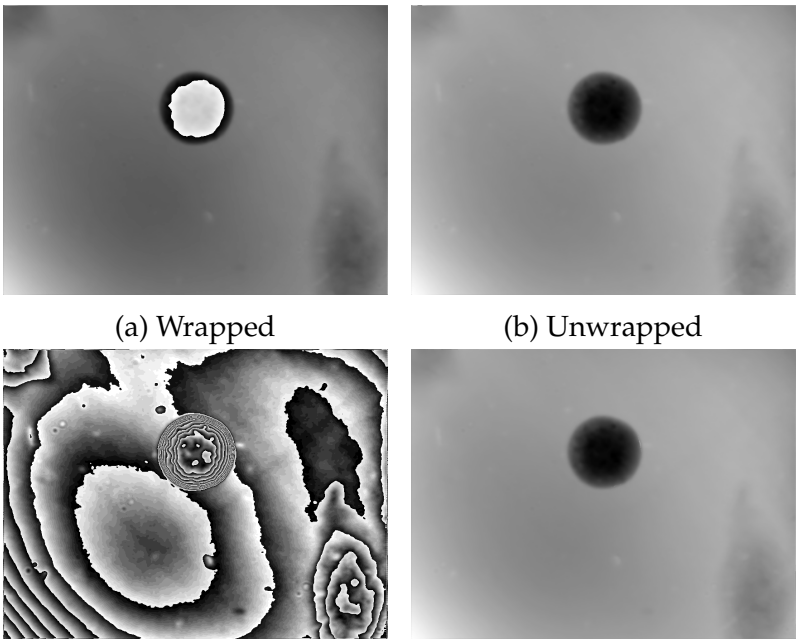
(c) Wrapped, factor 2 (d) Unwrapped, factor 2

Figure 4.2: Phase of a test image

in the tables displaying performance and is unnecessary if the CPU does not process the data. Some transfer is required by the residue pairing and flood-filling kernels, because they are run iteratively and the condition needs to be checked. The transferred data is only one 32-bit integer, but it prevents the algorithm to run asynchronously from the CPU.

4.5 Combination of GPU and CPU parts

Each component of the pipeline is composed of multiple parts, that correspond to kernels in CUDA. These parts were also implemented



(a) Wrapped (b) Unwrapped

(c) Wrapped, factor 15 (d) Unwrapped, factor 15

Figure 4.3: Image of a cell

on CPU, because on some images, it might by faster to use them than the GPU ones. For small instances of a problem CPU is usually faster, because GPU cannot be fully utilized.

For all of the images tested in this thesis, the GPU versions of all stages of the pipeline were significantly faster than the CPU versions, except for matrix decomposition and linear system solving. However, the time spent by computing the aforementioned tasks is negligible, compared to the rest of the pipeline. Hence, we did not combine the CPU and GPU methods together, as we consider the fully GPU-accelerated implementation close enough to optimal.

Table 4.6: Performance of polynomial fitting on the image in Figure 2.8

With the resolution 702 x 701.

	GPU Float	GPU Double	CPU Double
Memcpy to GPU	0.92 ms	1.64 ms	-
Reduction of coefficients	0.23 ms	3.88 ms	32.10 ms
Decomposition and solve	0.03 ms	0.05 ms	< 0.01 ms
Surface subtraction	0.06 ms	0.25 ms	5.51 ms
Memcpy from GPU	1.07 ms	2.07 ms	-
Total time	4.59 ms	11.42 ms	40.25 ms

Table 4.7: Performance of polynomial fitting on the image in Figure 4.3

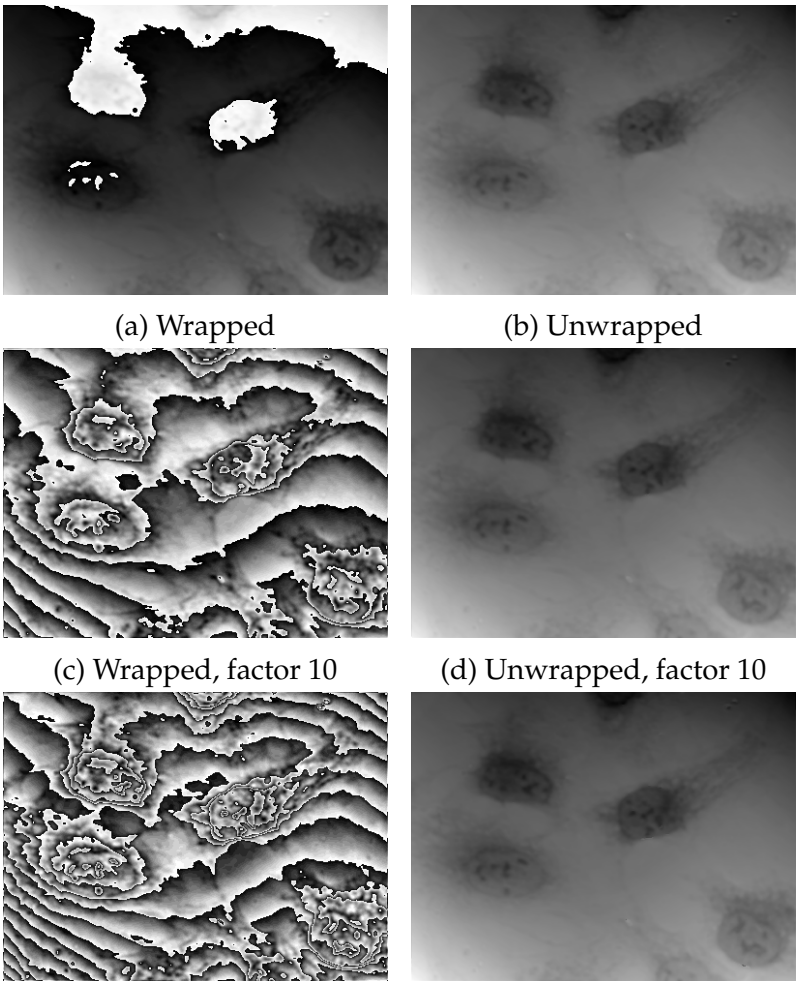
With the resolution 850 x 640.

	GPU Float	GPU Double	CPU Double
Memcpy to GPU	1.01 ms	1.82 ms	-
Reduction of coefficients	0.24 ms	4.07 ms	35.45 ms
Decomposition and solve	0.06 ms	0.04 ms	< 0.01 ms
Surface subtraction	0.07 ms	0.28 ms	6.11 ms
Memcpy from GPU	1.15 ms	2.28 ms	-
Total time	4.65 ms	12.50 ms	40.25 ms

Table 4.8: Performance of polynomial fitting on the image in Figure 4.4

With the resolution 398 x 299.

	GPU Float	GPU Double	CPU Double
Memcpy to GPU	0.25 ms	0.42 ms	-
Reduction of coefficients	0.13 ms	0.92 ms	7.75 ms
Decomposition and solve	0.03 ms	0.05 ms	< 0.01 ms
Surface subtraction	0.02 ms	0.07 ms	1.34 ms
Memcpy from GPU	0.28 ms	0.54 ms	-
Total time	1.16 ms	2.79 ms	9.68 ms

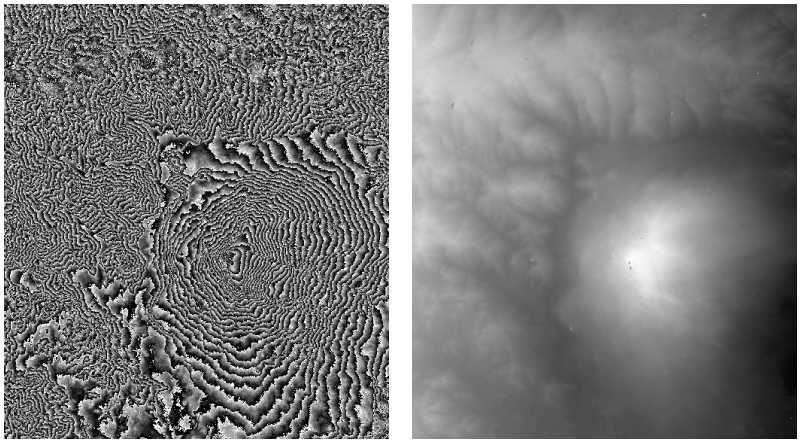


(a) Wrapped (b) Unwrapped

(c) Wrapped, factor 10 (d) Unwrapped, factor 10

(e) Wrapped, factor 15 (f) Unwrapped, factor 15

Figure 4.4: Image of cells

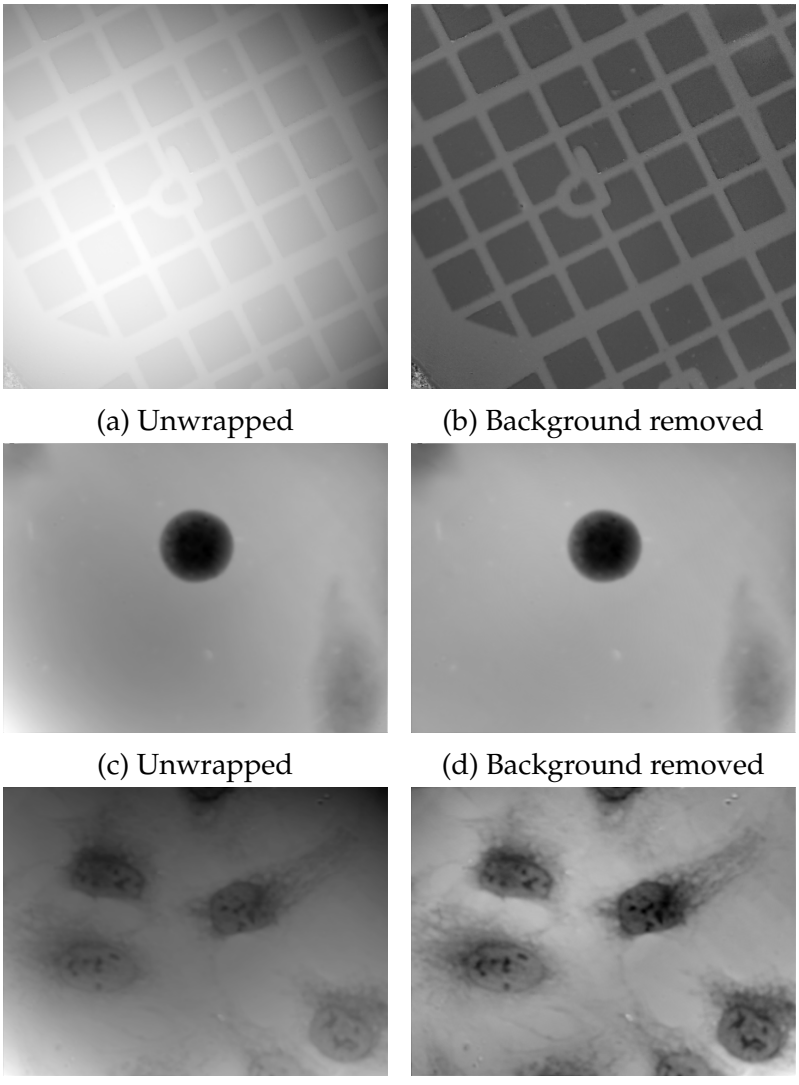


(a) Wrapped (b) Unwrapped

Figure 4.5: Image of a terrain from [19]

Table 4.9: Precision of the least squares method on GPU compared to the reference CPU implementation

	Float		Double	
	MSE	PSNR	MSE	PSNR
Figure 2.8	10^{-7}	114.44	10^{-17}	210.22
Figure 4.3	10^{-9}	135.10	10^{-18}	222.26
Figure 4.4	10^{-9}	135.87	10^{-22}	261.78
Figure 4.5	10^{-7}	110.60	10^{-17}	214.98



(a) Unwrapped (b) Background removed

(c) Unwrapped (d) Background removed

(e) Unwraped (f) Background removed

Figure 4.6: Images before and after background removal.

5 Conclusion

This thesis described some of the fundamental algorithms used in digital holographic microscopy. The algorithms were implemented and tested on both the CPU and GPU architecture.

On all the testing data, the GPU implementation was faster than the CPU implementation. On a typical image with the resolution around 400x400 and a small number of residue points, the performance of the GPU pipeline is approximately five times better than the CPU pipeline. On images with higher amount of noise or of larger resolution, the benefit of using the GPU was even more significant. The tables in chapter 4 show approximately ten times better performance. The precision of the GPU implementation is practically the same as the CPU reference implementation, the difference is negligible.

This work can be extended by further optimizing the kernels or by suppressing other artifacts in the image, mainly caused by the impurities in the optical system. It should be possible to more optimize the most time consuming part of the pipeline, the flood-fill integration and the residue pairing.

A Content of the digital archive

The attached archive contains the following items.

- `project/CudaPipeline` folder contains the implementation of the algorithms. To build the project the NetBeans IDE configured for remote building was used, but it should be straightforward to build it in other environments as well. All used libraries are included in the `include` and `lib` directories. The `CImg` library is used for loading and displaying images and the `newmat10` is used by the reference CPU implementation. The program takes two parameters the path to wrapped phase image and an optional image value coefficient.
- `project/CudaPipeline/imgs` contains images, which were used for testing the algorithms.
- `project/stats` directory contains the results of performance and precision tests.
- `thesis` directory contains the PDF version of this thesis and the LaTeX source used to create it.

Bibliography

[1] I. Moon, M. Daneshpanah, A. Anand, and B. Javidi, "Cell identification with computational 3-d holographic microscopy," June 2011. [Online] URL: http://www.osa-opn.org/home/articles/volume_22/issue_6/features/cell_identification_computational_3-d_holographic/.

[2] NVIDIA, *CUDA C Programming Guide*, October 2012. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[3] NVIDIA, *CUDA C Best Practices Guide*, October 2012. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[4] R. C. Gonzales and R. E. Woods, *Digital Image Processing*. Prentice Hall, 2nd ed., 2002.

[5] A. Oppenheim, R. Schafer, J. Buck, and et al, *Discrete-time signal processing*, vol. 2. Upper Saddle River, NJ: Prentice Hall, 1989.

[6] N. Petrovic, "Graphical models for 2d phase unwrapping." URL: http://www.ifp.illinois.edu/~nemanja/phase.html Accessed: 18.5.2013.

[7] S. Karout, *Two-Dimensional Phase Unwrapping*. PhD thesis, Liverpool John Moores University, April 2007. URL: http://www.ljmu.ac.uk/GERI/Theses/Salah_Karout_Thesis_.pdf.

[8] Z. N. Karam, "Computation of the one-dimensional unwrapped phase," Master's thesis, Massachusetts Institute of Technology, January 2006. URL: http://www.rle.mit.edu/dspg/documents/KaramMastersThesis.pdf.

[9] European Space Agency, "Phase unwrapping," 2007. URL: http://earth.esa.int/landtraining07/D1LB4-Rocca.pdf.

[10] R. M. Goldstein, H. A. Zebker, and C. L. Werner, "Satellite radar interferometry: Two-dimensional phase unwrapping," *Radio Science*, vol. 23, pp. 713–720, July - August 1988. URL: http://igppweb.ucsd.edu/~fialko/insar/Goldstein_RadioSci1988.pdf.

[11] E. W. Weisstein, "Least squares fitting–polynomial." URL: http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html Accessed 18.5.2013.

[12] C. D. Meyer, ed., *Matrix analysis and applied linear algebra*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000.

[13] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[14] NVIDIA, *CUFFT Library*, October 2012. URL: http://docs.nvidia.com/cuda/pdf/CUDA_CUFFT_Users_Guide.pdf.

[15] C. Flanagan, "The bresenham line-drawing algorithm." URL: http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html Accessed 18.5.2013.

[16] M. Harris, "Optimizing parallel reduction in CUDA." URL: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[17] R. Rahman, "The intel math kernel library and its fast fourier transform routines," 2011. URL: http://software.intel.com/en-us/articles/the-intel-math-kernel-library-and-its-fast-fourier-transform-routines.

[18] NVIDIA, "CUDA Fast Fourier Transform library." URL: https://developer.nvidia.com/cufft Accessed: 18.5.2013.

[19] M. Costantini, "A phase unwrapping method based on network programming," 1996. URL: http://earth.esa.int/workshops/fringe_1996/costanti/.