PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Nhan Hieu Do

Entitled
PARALLEL PROCESSING FOR ADAPTIVE OPTICS OPTICAL COHERENCE TOMOGRAPHY (AO-OCT) IMAGE REGISTRATION USING GPU

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

John J. Lee Chair	
Donald T. Miller	
Brian King	
Paul Salama	

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): John J. Lee

Approved by: Brian King 6/29/2016
Head of the Departmental Graduate Program Date

PARALLEL PROCESSING FOR ADAPTIVE OPTICS OPTICAL COHERENCE TOMOGRAPHY (AO-OCT) IMAGE REGISTRATION USING GPU

A Thesis

Submitted to the Faculty

of

Purdue University

by

Nhan Hieu Do

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

August 2016

Purdue University

Indianapolis, Indiana

This thesis is dedicated to my Family.

ACKNOWLEDGMENTS

I would like express my sincere gratitude to my major professor, Dr. John Lee for his great support, motivation, and guidance for more than two years. In addition, I really appreciate the collaboration with and support by Dr. Donald Miller, Dr. Kazuhiro Kurokawa, and Dr. Zhuolin Liu for this research. I also would like to thank Dr. Brian King and Dr. Paul Salama for their help and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

AO-OCT	Adaptive Optics Optical Coherence Tomography
CUDA	Compute Unified Device Architecture
GPGPU	General Purpose Computing on Graphics Processing Units
FFT	Fast Fourier Transform
IFFT	Inverse Fast Fourier Transform
DFT	Discrete Fourier Transform
IDFT	Inverse Discrete Fourier Transform
POC	Phase-Only Correlation
NCC	Normalized Cross-Correlation

ABSTRACT

Do, Nhan Hieu. M.S.E.C.E., Purdue University, August 2016. Parallel Processing for Adaptive Optics Optical Coherence Tomography (AO-OCT) Image Registration using GPU. Major Professor: John Jaehwan Lee.

Adaptive Optics Optical Coherence Tomography (AO-OCT) is a high-speed, high-resolution ophthalmic imaging technique offering detailed 3D analysis of retina structure in vivo. However, AO-OCT volume images are sensitive to involuntary eye movements that occur even during steady fixation and include tremor, drifts, and micro-saccades. To correct eye motion artifacts within a volume and to stabilize a sequence of volumes acquired of the same retina area, we propose a stripe-wise 3D image registration algorithm with phase correlation. In addition, using several ideas such as coarse-to-fine approach, spike noise filtering, pre-computation caching, and parallel processing on a GPU, our approach can register a volume of size $512 \times 512 \times 512$ in less than 6 seconds, which is a $33\times$ speedup as compared to an equivalent CPU version in MATLAB. Moreover, our 3D registration approach is reliable even in the presence of large motions (micro-saccades) that distort the volumes. Such motion was an obstacle for a previous *en face* approach based on 2D projected images. The thesis also investigates GPU implementations for 3D phase correlation and 2D normalized cross-correlation, which could be useful for other image processing algorithms.

1. INTRODUCTION

Adaptive Optics Optical Coherence Tomography (AO-OCT) is a high-speed, high-resolution ophthalmic imaging technique offering detailed 3D analysis of retina structure. The AO-OCT system in Dr. Miller's Lab at Indiana University Bloomington (IUB) allows acquisition of retinal volume images in vivo down to cellular level, which permits non-invasive visualization of retinal cells as for example cone photoreceptors [1]. Studying these volume images offers the promise of earlier detection of various eye-related conditions such as macular holes, retinal detachments, glaucoma, and age-related macular degeneration, all of which can lead to blindness [2–4]. This benefit, however, is not without cost. For a typical scanning session, AO-OCT systems generate enormous streams of raw data (streaming at 1.6 Gb/s), which quickly adds up to terabytes in size for a single patient. Moreover, these high resolution volume images are sensitive to eye motion artifacts because of involuntary eye movements during fixation including tremor, drifts, and micro-saccades [5]. As a result, even when we acquire a sequence of volume images of the same retina patch, each volume has its own unique pattern of distortions (caused by tremor and micro-saccades) and shifts (caused by drifts and head movement).

Therefore, analyzing retinal volume images requires motion correction using efficient and scalable algorithms. In addition to correcting intra-volume distortion, we also need to stabilize a sequence of volumes with 3D image registration. Given a sequence of volume images, 3D image registration will facilitate one-to-one mapping from the coordinates of thousands of A-lines in each volume to those of a chosen reference volume (chosen manually or automatically). Current approaches using MATLAB (CPU) take hours to register a set of volume images, which costs researchers' time and also makes it impossible to implement for real-time applications.

This thesis explores the possibility of using General Purpose Computing on Graphics Processing Units (GPGPU) to register retinal volume images with a reasonable speed and ideally, real time. With the power of parallel processing from thousands of cores, GPU is especially suitable for image registration algorithms with a high degree of parallelism, such as correlation-based registration algorithms. This thesis will present a stripe-wise registration method for AO-OCT volumes with 3D Phase-Only Correlation (3D POC) and Normalized Cross-Correlation (NCC). This approach will correct both axial and lateral directions. The registration accuracy and run-time will be evaluated quantitatively and qualitatively.

The organization of the thesis is as follows. Chapter 2 introduces AO-OCT technology and benefits of using image registration for reducing eye motion artifacts. Chapter 3 describes parallel computing technology with GPGPU. Chapter 4 gives an overview of correlation-based registration algorithms, including normalized cross-correlation and phase-only correlation. Chapters 5 and 6 explain GPU implementations of those two algorithms. Chapters 7 and 8 explain how to apply image registration algorithms to AO-OCT data for 3D registration along with our GPU implementation. Finally, Chapters 9 and 10 discuss our experimental results and summarize our contribution.

2. ADAPTIVE OPTICS OPTICAL COHERENCE TOMOGRAPHY

2.1 From OCT to AO-OCT

Since introduced in 1990, Optical Coherence Tomography (OCT) has utilized the principle of low coherence interferometry for micrometer-resolution visualization of biological tissue's internal structure in vivo [6]. This instrument is analogous to ultrasound imaging; however, the imaging beam is infra-red light instead of sound wave. Using OCT for retinal imaging, a researcher can construct 3D volume images by raster scanning an infra-red imaging beam across tissue. Over the last two decades, OCT technology has evolved from time-domain OCT (TD-OCT) to spectral-domain OCT (SD-OCT), which improved axial resolution from $10\mu m$ to $5\mu m$ while greatly reducing image acquisition time [7]. While these OCTs can provide such high axial resolution, their images still have low lateral resolution (over $15\mu m$) [8] . To improve lateral resolution, researchers successfully combined OCT with Adaptive Optics (AO) to create an AO-OCT system. Using a complex array of deformable mirrors, wave front sensors, high-speed cameras, lenses, etc., AO-OCT enabled researchers to drastically improve lateral resolution down to $3\mu m$ by correcting ocular aberrations [1]. When applied to ophthalmic imaging, AO-OCT systems combine the lateral resolution advantage of AO with the axial resolution advantage of OCT to provide an ultra-high 3D resolution of the retina, which is sufficient to visualize individual cells such as cone photoreceptors.

2.2 Eye Motion Artifacts and Image Registration

In Dr. Miller's Advanced Ophthalmic Imaging Lab at IUB, the research team has obtained retinal images with 3D resolution up to $3\mu m$ in each direction [9, 10] with scanning speed up to 1 MHz [1]. To describe these volume images, we use specific terminology as follows. In Figure 2.1, an A-line (amplitude scan) measures a light scattering intensity profile as a function of depth when the imaging beam scans axially to the tissue (z-direction). When the beam moves in the x-direction, a stack of A-lines constitutes a 2D image known as fast B-scan (brightness scan). The A-line intensity plot shows a specific depth profile of a specific A-line (in white) in the fast B-scan shown at the right side of the figure.

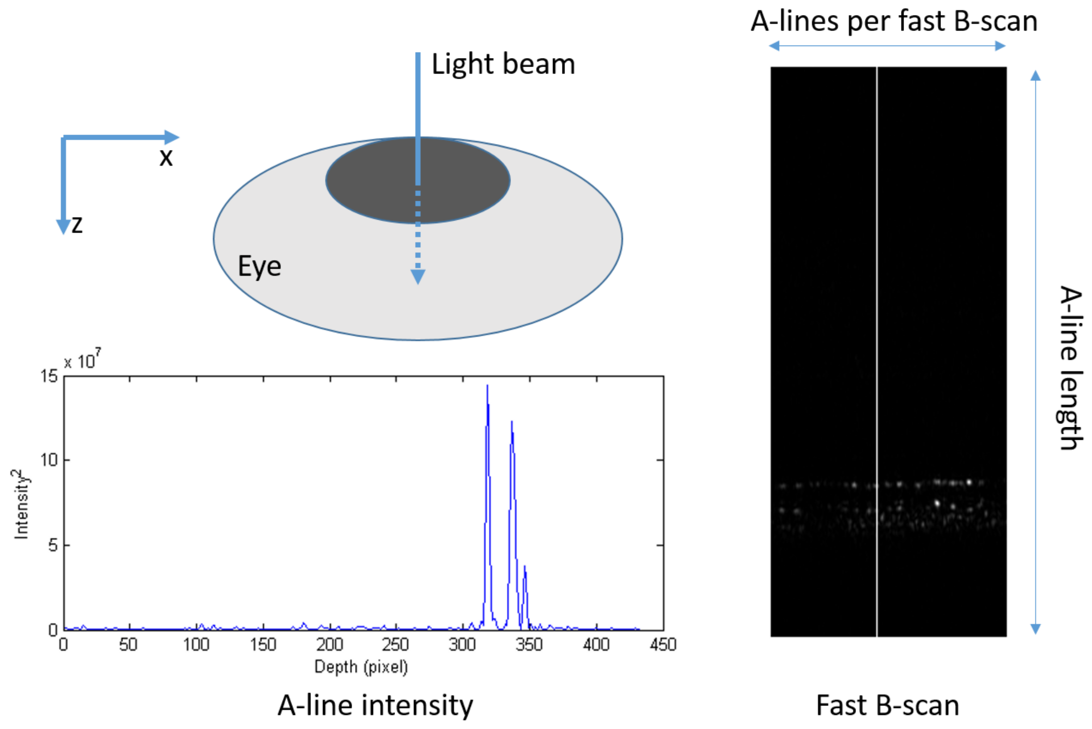


Fig. 2.1.: A-line and fast B-scan in an AO-OCT volume image.

In Figure 2.2, stacking fast B-scans in the y-direction will construct a volume image of a retina. Projecting a volume onto the yz-plane by averaging all pixels in the x-direction creates a slow B-scan projection, which shows different layers of cellular structures. The vertical irregular shift among fast B-scans appearing in a slow B-scan is caused by a subject's head movement in axial direction (inwards or outwards against the scanning instrument). Projecting a volume on the xy-plane by averaging all pixels in the z-direction creates a C-scan projection (or called *en face* projection). In high resolution, a C-scan can show cone cells and blood vessels. The distortion in the C-scan in Figure 2.2 is caused by involuntary eye motion under fixation on the xy-plane such as micro-saccade. We assume that eye motion is not significant within one fast B-scan because the acquisition time for a fast B-scan is so fast (1 to 2 ms) that the retina is considered to be stand still during that time. However, there are eye motions between different fast B-scans.

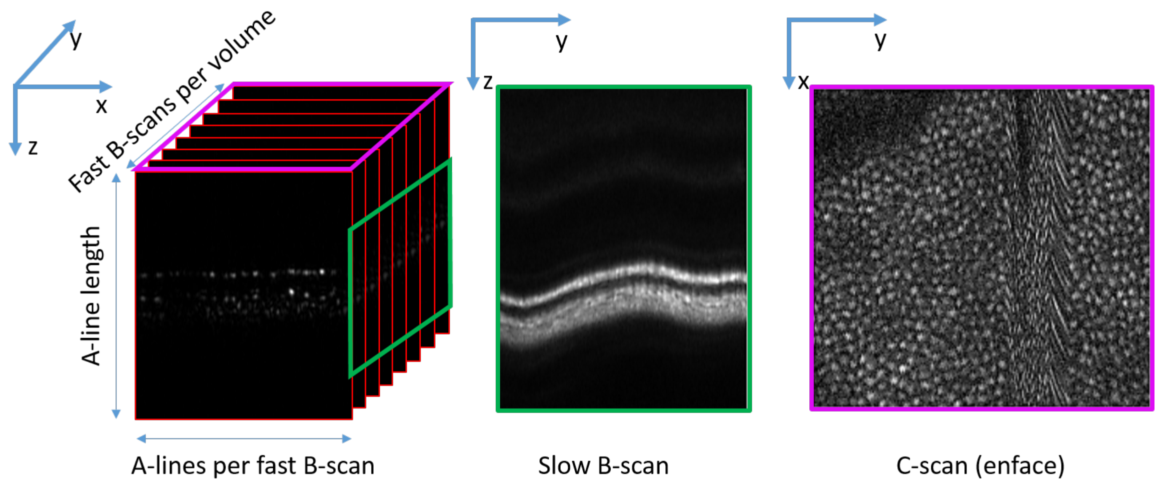


Fig. 2.2.: Slow B-scan and C-scan in AOOCT volume image.

Obtaining a sequence of volume images creates a time-series 3D image of ideally the same retinal patch. However, because eye movements occur as the imaging beam scans, the acquired volumes are distorted relatively to one another. Figure 2.3 illustrates the lateral movement in a form of C-scan projection between a target volume and a reference volume. In a series of volume images, one volume with the highest quality (low distortion and high contrast) is chosen to be the reference volume. The task of image registration is to map the coordinates of A-lines from other volumes (which we call target volumes) to the coordinate of the reference volume. The output of a registration algorithm is 3D displacement of each A-line in the x, y and z directions relative to the corresponding A-line in the reference volume.

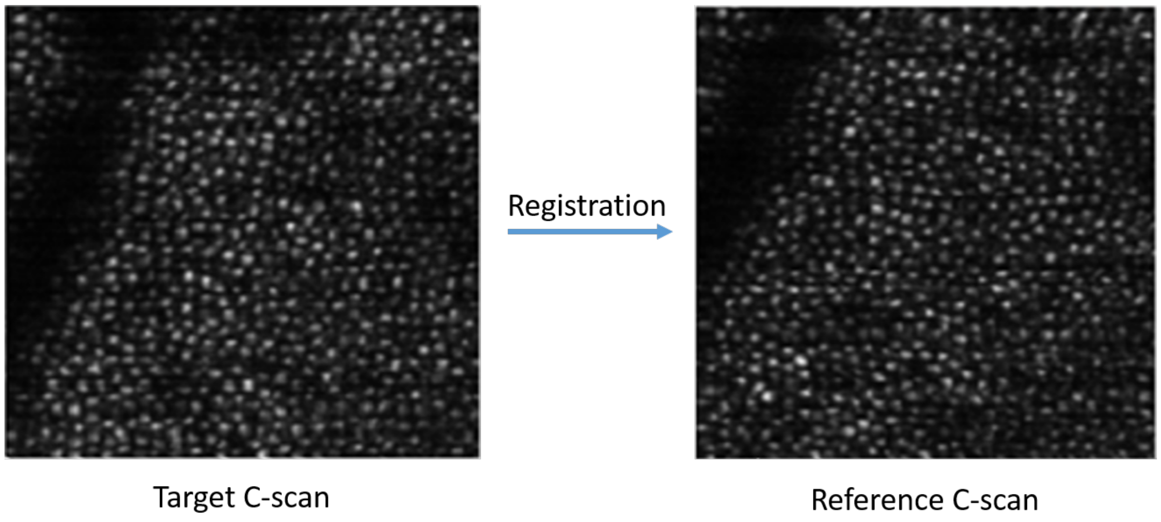


Fig. 2.3.: Lateral movement between a target image and a reference image.

Effective image registration can provide many benefits. For on-the-fly processing, detecting eye motion in real time can provide feedback for the system for eye tracking and maintaining the field of view. For post-processing, as we obtain many volumes of the same retina area, we can combine data from many low-quality volumes to create a high-quality volume with improved signal-to-noise ratio as well as to correct ocular image distortion. In addition, we can also analyze data from the same retina area for an extended period of time such as days or months to observe cellular-level changes, for example, the development of an eye-related disease [10].

3. PARALLEL COMPUTING WITH GPGPU

3.1 The Rise of Parallel Computing

Conventionally, computer programs run in a serial manner, meaning that each instruction is executed one after another using a CPU (Central Processing Unit). Therefore, speed of execution highly depends on CPU's frequency (clock rate). Thus, to improve performance, the industry has eventually increased CPU's clock rate: from 2 MHz for the Intel 8080 CPU in 1975 to about 3-4 GHz for modern CPUs such as the Intel Core i7 processor. However, modern CPU cores are reaching the physical limitation on transistors' size and heat restriction. As a result, instead of containing a single core, a processor contains many cores on a chip (two, four, eight, or sixteen cores), with the goal of increasing throughput by having many cores process data independently in parallel.

The trend of more fine-grained parallel computing has started in 2010 [11]. Beginning in the 20^{th} century, Graphics Processing Units (GPUs) gained attraction with a different processing architecture. Instead of having a few powerful cores like a CPU, a GPU employs hundreds to thousands of simple cores for massively parallel computing. At first, GPUs were used for graphics rendering and pixel shading, and were tricky to be used for other computing tasks. To support general purpose computing, in November 2006, NVIDIA released GeForce 8800 GTX series, which was the first GPU with CUDA architecture, effectively creating the first GPGPU (General Purpose computing on Graphics Processing Unit). Since then, GPUs have been used for high performance computing for various applications with a large amount of data especially in science and engineering.

3.2 Compute Unified Device Architecture (CUDA)

CUDA is an architecture designed to enable developers to use a GPU for general purpose computing tasks. The language for CUDA is very similar to C with additional keywords for utilizing GPU resources. With CUDA, industries have improved performance of previous algorithm implementations by orders-of-magnitude speedup [11]. At the time of writing, NVIDIA has released CUDA 7.5 with extra features to allow flexible programming design with support for C++11, which was not possible in previous CUDA.

An application written for GPGPU includes host code and device code. Host code refers to the part of application that executes on a CPU and accesses main memory such as system hard disk or RAM. On the other side, device code refers to the part of application that runs on a GPU and has access to GPU's memory. Device code is organized into kernels. They are functions that allow passing arguments such as configuration parameters or pointers to input and output data. A kernel can be invoked by either from host code or from other kernels.

When launched, the same kernel will be executed by thousands of threads concurrently. Threads are grouped into blocks, and blocks are grouped into a grid. Each thread or block has an identifier such as *threadIdx* or *blockIdx*. Using these identifiers and the information about block dimension *blockDim* (how many threads in a block) and grid dimension *gridDim* (how many blocks in a grid), each thread can calculate the unique address of its input or output memory. The block dimension and grid dimension are chosen such that there are enough threads to share the total workload and to be able to hide the memory latency.

The memory hierarchy of a GPU contains many types: registers, shared memory, texture memory, constant memory, and global memory, in the order of increasing latency [12]. Each thread has its private registers with a latency to read/write is one or a few clock cycles. Threads in the same block can communicate via shared memory with a reasonable latency (1 - 32 clock cycles). All threads can access to texture memory, constant memory and global memory, however, with a latency of up to 400 - 600 clock cycles. Although registers and shared memory are fast, their sizes are very limited compared to global memory. Nevertheless, latency to global memory can be reduced with caches such as L1 cache and L2 cache depending on GPU type.

Therefore, accessing memory effectively is crucial for GPU programming because many applications are not arithmetically bounded but are memory bandwidth bounded. This means that an application should utilize registers and shared memory as much as possible before considering global memory. When reading/writing to global memory is required, some techniques to increase throughput include coalesced memory access, usage of constant memory and texture memory, and GPU streams. Memory coalescing allows combining memory accesses from multiple threads in one transaction. Constant memory and texture memory are just special areas of global memory but are cached. GPU streams allow overlapping data transfer and kernel execution, thus increasing throughput.

3.3 Maxwell Architecture

The GPU micro-architecture has evolved for many generations: G70, Tesla, Fermi, Kepler, and most recently, Maxwell (developed in 2014). Maxwell introduces many advancements over previous architectures and allows better workload balancing, instruction scheduling, and power efficiency. As an improvement from SMX (Kepler streaming processor), Maxwell employs the new SMM (Maxwell streaming processors) that has four 32-core processing blocks, eight texture units, and one PolyMorph Engine (Figure 3.1). The size of shared memory is also larger and more dedicated. In Fermi and Kepler, each SMX has 64 KB of fast memory to be partitioned between shared memory and L1 cache. In Maxwell, as L1 and texture caches are combined into a single unit, each SMM now has whole 64 KB dedicated shared memory, and even up to 96 KB for the second generation of Maxwell. This large shared memory will allow more blocks to occupy the SMM concurrently [13].

In this research, we utilize an NVIDIA Titan X GPU, which has the second generation of Maxwell architecture. With 3072 CUDA cores, it provides an increased performance for single-precision floating point computation (6 TFLOPS). Titan X also has 12 GB of GDDR5 RAM with memory bandwidth up to 336.5 GB/s [14] that allows us to process large volume images. However, Titan X has a poor double-precision floating point computation performance (0.19 TFLOPS) [15]. Therefore, the floating point computation of our algorithms should better be single-precision. However, float overflow and underflow may happen during computation and should be handled.

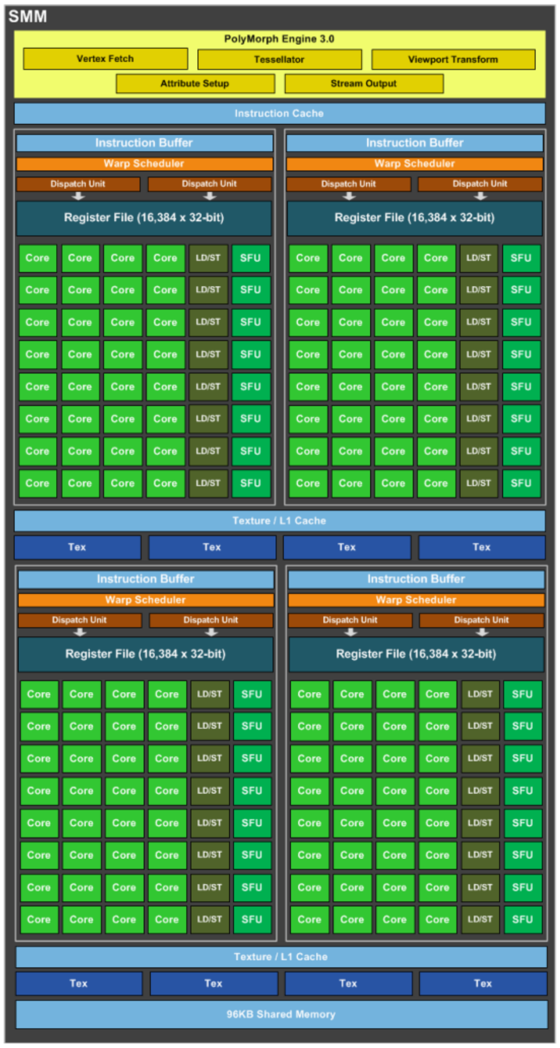


Fig. 3.1.: Maxwell SMM streaming processor [13].

4. IMAGE REGISTRATION WITH CORRELATION METHODS

4.1 Image Registration Theory

The purpose of image registration is to align images of the same object that are somewhat different due to distortions, movements, and differences in acquisition angles, time, and/or sensors used. This is an important step to process data before further analysis in many areas, especially in medical imaging. The image that we consider to be stationary is called a reference image, while those moving relatively to the reference image are called target images.

There are two main categories of image registration: feature-based and correlation-based. In a feature-based method, common features between a reference image and a target image are identified and then a point-by-point mapping between them is estimated. The feature selection can be manual, semi-automatic, or fully automatic. For 2D OCT images, features can be blood vessels or the optic disc. For 3D AO-OCT data, features can be blood vessels and cones. However, because AO-OCT volume images zoom in a small area on a retina, it is harder to find prominent features because the optic disc is missing and blood vessels are few. In addition, it is hard to establish an automatic 3D feature selection algorithm. Alternatively, a correlation-based method will match image textures between a reference image and a target image. This can be done either in the spatial domain with Normalized Cross-Correlation (NCC) [16] or in the frequency domain with Phase-Only Correlation (POC), or Phase Correlation in short [17].

4.2 Normalized Cross-Correlation

NCC is frequently used for template matching, in other words, finding a sub-image in a reference image that provides the best match for a chosen template. Let $t(x, y)$ be a template image and $f(x, y)$ be a sub-image that has the same size as the template image. Then, NCC between them is calculated as follows:

$$\frac{1}{N}\sum_{x,y}\frac{(f(x,y)-\mu_f)(t(x,y)-\mu_t)}{\sigma_f\sigma_t} \tag{4.1}$$

In this equation, μ_f and μ_t are averaged pixel intensity values (means) of f and t; σ_f and σ_t are standard deviations of f and t; and N is the number of pixels in f. Note that in Equation 4.1, each image is subtracted by its mean and then divided by its standard deviation before calculating cross-correlation. This normalization step can improve accuracy in template matching in case f and t have different brightness levels.

Next, a correlation map can be obtained by calculating NCC for all possible sub-images in the reference image by moving a template image over the entire reference image. This process is similar to a convolution between two images. Therefore, if a template image has size M and a reference image has size N, the correlation map will have size $M + N - 1$. The largest correlation coefficient value in this map denotes the best match position, which is also the translationally shifted amount between two images.

4.3 Phase Correlation

The following section will demonstrate image registration using POC between two 2D images. However, the formula can be generalized for higher dimension cases such as 3D. We only consider a translational shift with no rotation or scale between those two images in our study.

Let $f_a(x, y)$ be a reference image and $f_b(x, y)$ be a target image. Assume that the target image is a translationally shifted version of the reference image by $(\Delta x, \Delta y)$, then:

$$f_b(x, y) = f_a(x - \Delta x, y - \Delta y) \tag{4.2}$$

Applying Fourier Transform (FT) to both sides of the equation, we obtain F_a and F_b that are the FT of f_a and f_b, respectively. Here we apply the Fourier Shift Theorem, which states that a linear shift in the spatial domain will be equivalent to a phase shift in the frequency domain as follows:

$$F_b(u, v) = F_a(u, v)e^{-j(u\Delta x + v\Delta y)} \tag{4.3}$$

Then we can calculate cross-power spectrum by taking point-wise multiplication between F_a and the complex conjugate of F_b (denoted by F_b^*) and normalize the product to keep only the phase information in the exponential.

$$\frac{F_a(u, v)F_b^*(u, v)}{|F_a(u, v)F_b^*(u, v)|} = e^{j(u\Delta x + v\Delta y)} \tag{4.4}$$

The Inverse Fourier Transform (IFT) of the cross-power spectrum will result in a Dirac delta function δ centered at $(-\Delta x, -\Delta y)$ as follows:

$$\mathcal{F}^{-1}\{e^{j(u\Delta x + v\Delta y)}\} = \delta(x + \Delta x, y + \Delta y) \tag{4.5}$$

In practice, the shift $(\Delta x, \Delta y)$ may not have integer values. As a result, phase correlation can even determine a sub-pixel level shift [18, 19] based on different interpolation methods. Phase correlation can also be extended to find rotation and scaling differences [17]. However, our application only requires an estimated rigid translational pixel level shifts, and thus, we do not consider those extensions, which are more complicated to implement.

Figure 4.1 compares using a normalized cross-correlation method and a phase correlation method to register between a reference image and its shifted version as a target image. The target image is shifted by $(\Delta x, \Delta y) = (-50, -30)$. By finding the peak in the phase correlation from Figure 4.1(d), we can estimate the translational shift between the target image and the reference image.

Figure 4.1(c) shows the correlation map if we use NCC. This map is cropped to have the same size with the reference image. The peak showing the highest correlation value is more apparent in POC than in NCC. In addition, the execution time of NCC takes longer than that of POC. For the given example in Figure 4.1, MATLAB takes 5 ms for POC whereas 52 ms for NCC. Given that our application requires fast running time for large data size in 3D, we decide to base our approach on POC. However, NCC can be used as an alternative because NCC is more suitable for pixel level registration than POC. The reason is that NCC gives the best match position when moving a template image across a reference image one pixel at a time (template matching).

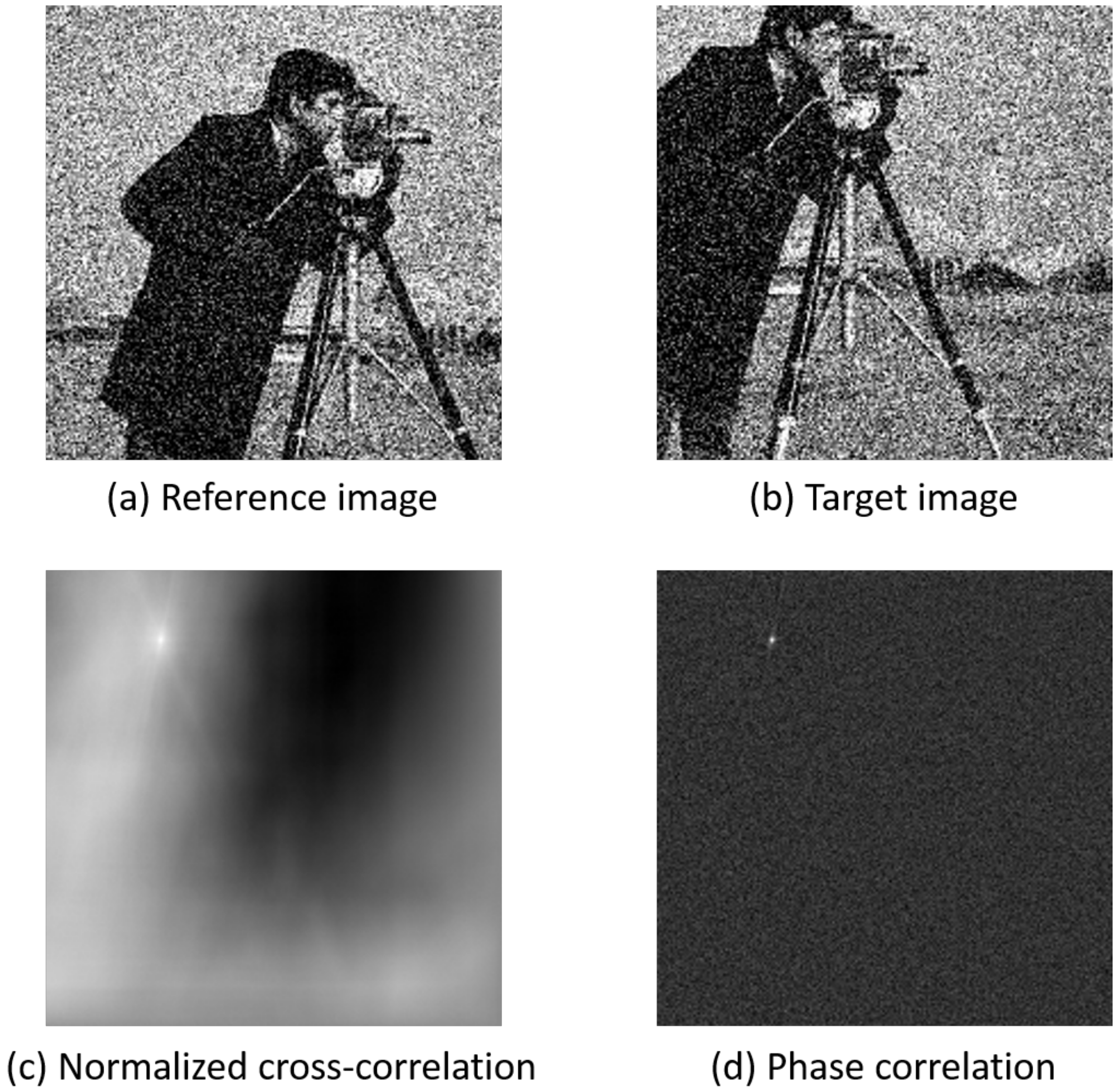


Fig. 4.1.: Demonstration of normalized cross-correlation versus phase correlation on two images contaminated with white Gaussian noise. (a) the reference image. (b) the target image. (c) the correlation map with normalized cross-correlation. (d) the correlation map with phase correlation.

5. 3D PHASE CORRELATION IN GPU

5.1 Implementation Overview

Following the phase correlation algorithm from Section 4.3, we designed a GPU program with the help of cuFFT library from NVIDIA. This library is built based on the popular FFTW library to do Fast Fourier Transform (FFT) on GPU [20]. Algorithm 1 summarizes our implementation.

Note that for the phase correlation computation, a target volume and a reference volume must have the same size. If their sizes are different, zero-padding should be done for the smaller volume before calling *PhaseCorrelation*.

To prepare for FFT, Lines 1 to 3 create a configuration (which is called plan) to optimize FFT for a given input size and the selected GPU hardware [20]. *cufftPlan3d* allows us to create plans for FFT in 1D, 2D, or 3D with either forward Discrete Fourier Transform (DFT) or backward Inverse Discrete Fourier Transform (IDFT). The equation for the forward DFT we use is as follows:

$$Y[u,v,w] = \sum_{x=0}^{NX-1} \sum_{y=0}^{NY-1} \sum_{z=0}^{NZ-1} X[x,y,z] e^{\frac{-2\pi j(ux)}{NX}} e^{\frac{-2\pi j(vy)}{NY}} e^{\frac{-2\pi j(wz)}{NZ}} \tag{5.1}$$

For the backward IDFT we use:

$$Y[u,v,w] = \frac{1}{NX \times NY \times NZ} \sum_{x=0}^{NX-1} \sum_{y=0}^{NY-1} \sum_{z=0}^{NZ-1} X[x,y,z] e^{\frac{2\pi j(ux)}{NX}} e^{\frac{2\pi j(vy)}{NY}} e^{\frac{2\pi j(wz)}{NZ}} \tag{5.2}$$

In both of the equations above, X is the input and Y is the output after the corresponding transformation.

Our phase correlation algorithm requires two DFTs and one IDFT. Instead of creating three plans, one for each of the transformations, we only need two plans. The forward plan is used for the two DFTs of the target volume and the reference volume because both volumes are assumed to have the same size. The backward plan

Algorithm 1 PhaseCorrelation ($tarVol, refVol$)

Note: $tarVol$ refers to a target volume and $refVol$ refers to a reference volume. We assume that both $tarVol$ and $refVol$ are of the same size (NX, NY, NZ). The output is the shifted amount $(\Delta x, \Delta y, \Delta z)$ of $tarVol$ in relative to $refVol$.

```
1: cufftHandle $fwplan, bwplan$
2: cufftPlan3d (&$fwplan$, $NX, NY, NZ$, CUFFT_R2C)
3: cufftPlan3d (&$bwplan$, $NX, NY, NZ$, CUFFT_C2R)

4: $realSize \leftarrow NX \times NY \times NZ$
5: $complexSize \leftarrow NX \times NY \times (NZ/2 + 1)$

6: cufftComplex* $tarVolF, refVolF$
7: cudaMalloc(&$tarVolF, complexSize \times sizeof(cufftComplex)$)
8: cudaMalloc(&$tarVolF, complexSize \times sizeof(cufftComplex)$)
9: cufftExecR2C ($fwplan, tarVol, tarVolF$)
10: cufftExecR2C ($fwplan, refVol, refVolF$)

11: $tarVolF \leftarrow$ PointwiseConjugateProduct ($refVolF$, $tarVolF$, $complexSize$)
12: cufftReal* $coefMap \leftarrow tarVol$
13: cufftExecC2R ($bwplan, tarVolF, coefMap$)
14: $(maxIndex, maxValue) \leftarrow$ FindMaxIndex ($coefMap, realSize$)
15: $(\Delta x, \Delta y, \Delta z) \leftarrow$ Ind2Sub ($maxIndex, NX, NY, NZ$)
16: $correlationCoef \leftarrow maxValue/realSize$
17: **return** $(\Delta x, \Delta y, \Delta z)$
```

is used for the IDFT. We name the forward plan as $fwplan$ and the backward plan as $bwplan$.

As mentioned in [20], when the input volume has real values like in our application, its DFT will have complex values and will satisfy Hermitian symmetry. This means that the output can be stored using almost half of the required memory space. Similar property also holds for Complex to Real IDFT. That is, if input data are in complex values and thus satisfy Hermitian symmetry, the output is purely real-valued. We show data sizes for these two transformations in Table 5.1.

Table 5.1: Data size in different 3D transformations in cuFFT

FFT type for 3D	Input data size	Output data size
Real to Complex (CUFFT_R2C)	$NX \times NY \times NZ$ cufftReal	$NX \times NY \times (\lfloor NZ/2 \rfloor + 1)$ cufftComplex
Complex to Real (CUFFT_C2R)	$NX \times NY \times (\lfloor NZ/2 \rfloor + 1)$ cufftComplex	$NX \times NY \times NZ$ cufftReal

Lines 4 and 5 in Algorithm 1 calculate the $realSize$ and $complexSize$. Next, Lines 6 to 8 allocate GPU memory for the output arrays for forward DFTs ($complexSize$ complex numbers per array). Complex number type is represented by the data type *cufftComplex* from cuFFT library.

Lines 9 and 10 perform the forward DFT. The function *cufftExecR2C* is part of the cuFFT library. It takes three parameters: a plan to execute, input, and output.

Line 11 calculates the normalized cross-power spectrum between *refVolF* and *tarVolF* as shown in the left hand side of Equation 4.4. The output is written to *tarVolF* to save memory space because we no longer need the DFT result of *tarVol* any more. The details of function *PointwiseConjugateProduct* will be discussed in Algorithm 2.

In Line 12, we use the memory allocated by *tarVol* to hold the correlation map because the data in *tarVol* is no longer needed. If we wish not to overwrite *tarVol*, extra memory of size *realSize* must be allocated using *cudaMalloc* for *coefMap*. Line 13 performs the backward IDFT with the input as our normalized cross-power spectrum (now stored in *tarVolF*). The output will be stored in a 3D matrix *coefMap*. Note that *bwplan* is used here.

Next, Line 14 finds the maximum element in *coefMap* using function *FindMaxIndex*. The details of this function will be discussed in Algorithms 3 and 4. The *maxIndex* returned from this function is the linear index of *coefMap*. As a consequence, function *Ind2Sub* is used in Line 15 to convert this linear index to 3D. The details of Line 15 will be discussed in Algorithm 5. This is the 3D shift amount between *tarVol* and *refVol* that needs to be returned as $(\Delta x, \Delta y, \Delta z)$.

In Line 16, we obtain the value of the maximum correlation coefficient directly from *maxValue*. Note that we have to normalize it by *realSize*. The reason is that for the backward IDFT, cuFFT only calculates the triple sum in Equation 5.2. Therefore, we have to divide the result by the total number of elements. Although we do not require this correlation coefficient value in our application, it is presented here for completeness.

5.2 Supplementary Kernels

5.2.1 Pointwise Conjugate Product

This function takes two arrays of complex numbers (A and B) and calculates $\frac{A \circ B^*}{|A \circ B^*|}$, where $\circ$ represents the point-wise product (Hadamard product) and B^* represents the complex conjugate of B. The result of the computation will overwrite data in B.

Lines 1 to Line 3 specify the number of threads per block and the number of blocks to be called when launching the Pointwise-Conjugate-Product-Kernel kernel in Line 4. We provide an option to limit the number of maximum blocks launched on the GPU (*MAXBLOCKS*) to provide flexibility for different GPUs. One can tune this parameter for better performance and utilization balance, for example, selecting *MAXBLOCKS* to be a multiple of the total number of SMs on a device.

Pointwise-Conjugate-Product-Kernel implements a grid-stride loop to enable threads reuse and scalability [21]. With thread reuse, the cost of construction and destruction of threads, along with the computation of *gtid* and *gridSize*

Algorithm 2 PointwiseConjugateProduct $(A, B, size)$

Note: Two complex array A and B are assumed to be the same size. The output will be overwritten to B. The type of A and B is *cufftComplex*. For a number a of this type, we can access the real part by $a.x$ and the complex part by $a.y$

```
1: $threads \leftarrow 256$
2: $MAXBLOCKS \leftarrow 1024$
3: $blocks \leftarrow min(\lceil size/threads \rceil, MAXBLOCKS)$
4: Pointwise-Conjugate-Product-Kernel $\lll blocks, threads \ggg (A, B, size)$    ▷ Call the GPU kernel

5: function Pointwise-Conjugate-Product-Kernel$(A, B, size)$
6:     $gtid \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
7:     $gridSize \leftarrow blockDim.x \times gridDim.x$
8:     while $gtid < size$ do    ▷ grid stride loop
9:         $a \leftarrow A[gtid].x$
10:        $b \leftarrow A[gtid].y$
11:        $c \leftarrow B[gtid].x$
12:        $d \leftarrow B[gtid].y$

13:        $retX \leftarrow a \times c + b \times d$
14:        $retY \leftarrow -a \times d + b \times c$
15:        $scale = 1/\sqrt{retX^2 + retY^2}$
16:        $B[gtid].x \leftarrow scale \times retX$
17:        $B[gtid].y \leftarrow scale \times retY$
18:        $gtid \leftarrow gtid + gridSize$    ▷ index update for grid stride loop
19:    end while
20: end function
```

before the loop in Line 8, is amortized over loop iterations. In addition, the grid-stride loop allows our kernel to run with any data size without worrying about the maximum number of blocks supported by the GPU.

For each iteration in the loop in Line 8, let $\alpha = A[gtid] = a+bj$ and $\beta = B[gtid] = c+dj$; then we have $\alpha\beta^* = (a+bj)(c-dj) = (ac+bd)+(-ad+bc)j$ (Lines 9 to 14). We then normalize this result by the Euclidean norm $\| \, (ac+bd)+(-ad+bc)j \, \|$ and save the result to an output array (overwritten to B in this case) (Lines 15 to 17). The normalization step is required to keep only the phase correlation information. After that, in Line 18, the *gtid* is incremented by a grid-stride and ready to calculate the next group of elements. Note that we check if the *gtid* is within the boundary of *size* in Line 8 to avoid illegal memory access.

5.2.2 Find the Index of the Maximum Element

This function takes an array of real numbers (A) and find the maximum value along with its linear index. Algorithm 3 demonstrates the process in two steps: Find-Max-Index-Kernel (shown in Algorithm 4) in the GPU side and Find-Max-Index-CPU in the CPU side.

Figure 5.1 demonstrates these two steps. The first step performs a well-known technique in GPU called reduction. For an array that has N elements, we can divide it into M blocks. Instead of sequentially checking each element in an array to find the maximum value, which requires $\mathcal{O}(N)$ steps, reduction performs comparisons in parallel across each block and reduces the number of steps needed per block down to $\mathcal{O}(log_2 N/M)$. The output will be stored in *outIndex* and *outArray*. Each element in the array holds the maximum value and its index for each block. As a result, the size of *outIndex* and *outArray* is M elements. For implementation, we adapt the optimized reduction code from [22], whose purpose is for summation reduction. Therefore, we modify it into maximum reduction with extra shared memory to keep track of the index of the maximum element. The process for maximum reduction

Algorithm 3 FindMaxIndex $(A, size)$

Note: Input is a real-value array A with the specified size. The output will be the index and value of the maximum element.

```
1: $threads \leftarrow 256$
2: $MAXBLOCKS \leftarrow 1024$
3: $blocks \leftarrow min(\lceil \text{size / threads} \rceil, MAXBLOCKS)$
4: $smemSize \leftarrow 2 \times threads \times sizeof(float)$
5: FIND-MAX-INDEX-KERNEL $\lll blocks, threads, smemSize \ggg$ $(A,$ $size,$
   $outIndex,$ $outData)$                                   ▷ Call the GPU kernel
6: return FIND-MAX-INDEX-CPU$(outIndex, outData, blocks)$

7: function FIND-MAX-INDEX-CPU$(indexArray, dataArray, size)$
8:     $maxValue \leftarrow$ maximum element in $dataArray$
9:     $maxIndex \leftarrow$ the corresponding index in $indexArray$
10:    return $(maxIndex, maxValue)$
11: end function
```

is illustrated in Figure 5.2. For the second step of reduction, we can either launch another GPU kernel to find the final maximum value or copy the data to the CPU and run the reduction on the host. We decide to choose the later approach because it would be wasteful to use the GPU for a small data size of M elements.

Lines 1 to 4 specify the parameters to launch the FIND-MAX-INDEX-KERNEL kernel. Note that along with the number of threads and the number of blocks, we have an extra parameter here for the size of the shared memory. In CUDA, the shared memory can be declared in the kernel itself, or dynamically declared in the kernel call (Line 5). The benefit of dynamic allocation is that we can specify the size of this memory at run-time rather than hard-coded in the kernel. In addition, note that we require that this shared memory is twice the number of threads per block (Line

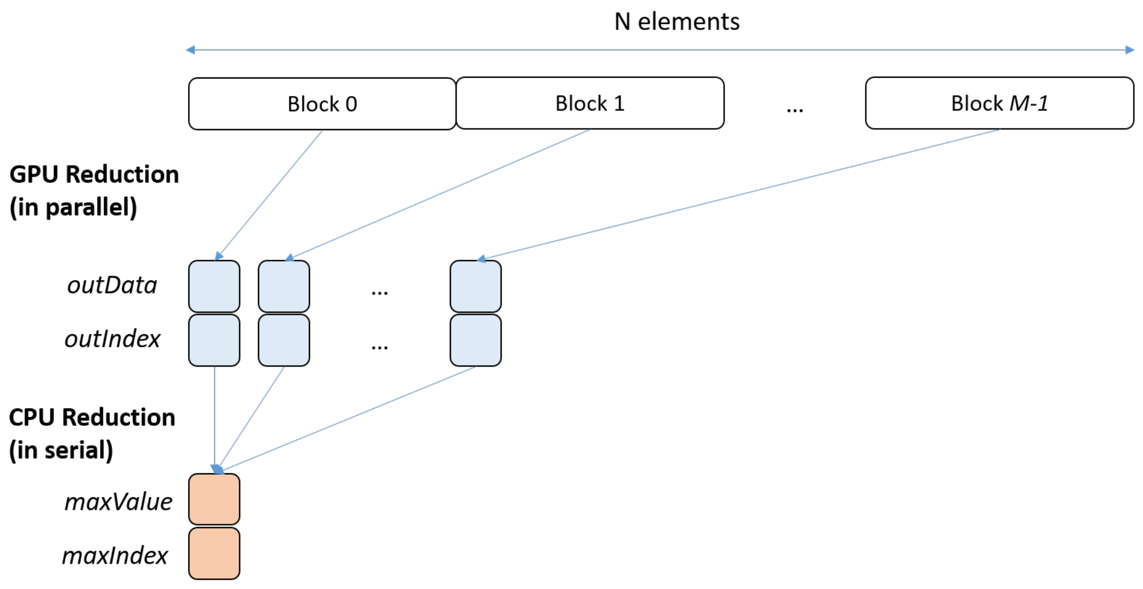


Fig. 5.1.: Two-step reduction in FindMaxIndex.

4) because we need to not only store the data values but also their corresponding indexes as well.

The detail of FIND-MAX-INDEX-KERNEL in Algorithm 4 is as follows. First, Lines 2 to 4 calculate the local/global thread indexes and the grid size. After that, Lines 5 through 9 declare and initialize the shared memory. Here we have two memory pointers $smem$ and $smemIndex$ to divide our shared memory to two halves of length $blockDim.x$ each: the first half is for $smem$ and the second half is for $smemIndex$. In addition, we need to declare the shared memories as volatile in Lines 6 and 7 to prevent incorrect behaviors caused by compiler optimization. Lines 10 to 14 load the data from global memory to the shared memory. Again, the grid stride loop enables us to reduce several elements per thread and increase the work load for each thread (technically called instruction level parallelism). Line 16 synchronizes the threads. It is necessary to avoid race condition because the next part requires the cooperation between threads using shared memory. In GPU, function __syncthreads() guarantees that all threads have finished writing to shared memory before continuing. Lines 17 through 22 perform the reduction in shared memory and cut the data by half for

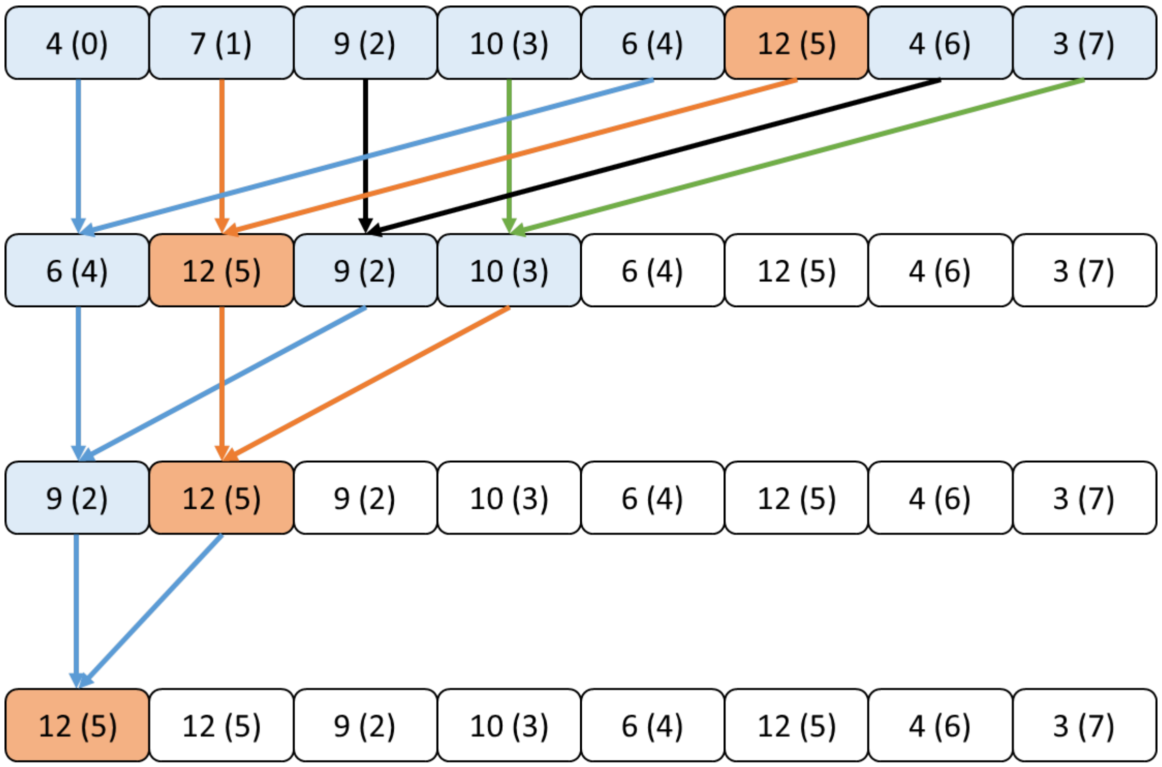


Fig. 5.2.: Reduction in GPU to find the maximum value and the maximum index. For a box that says $a(b)$, then a is the value and b is the corresponding index. From the original array in the first row, the maximum value is 12 at index 5.

each iteration. __syncthreads() is needed here because the threads need to wait for other threads to finish current step. However, when the number of active threads goes down to the warp size of 32 (Lines 23 - 30), we do not need __syncthreads() any more because threads within a warp are implicitly synchronized. Finally, the first thread in each block will output the reduction result for that block (Lines 31 - 34).

5.2.3 Convert Linear Index to 3D Subscript

This function converts from linear indexes to equivalent 3D coordinates (or called subscripts) of a 3D matrix. This is the implementation of a similar function in

Algorithm 4 Find-Max-Index-Kernel($A, size, outIndex, outData$)

```
1:  function Find-Max-Index-Kernel($A, size, outIndex, outData$)
2:      $tid \leftarrow threadIdx.x$
3:      $gtid \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
4:      $gridSize \leftarrow blockDim.x \times gridDim.x$
5:      extern __shared__ float $S[]$
6:      volatile float* $smem \leftarrow S$
7:      volatile int* $smemIndex \leftarrow \&smem[blockDim.x]$
8:      $smem[tid] \leftarrow FTL\_MIN$                    ▷ Smallest float number in GPU
9:      $smemIndex[tid] \leftarrow -1$
10:     while $gtid < size$ do                                 ▷ grid stride loop
11:         if $A[gtid] > smem[tid]$ then
12:             $smem[tid] = A[gtid]$, $smemIndex[tid] = gtid$
13:         end if
14:         $gtid \leftarrow gtid + gridSize$          ▷ index update for grid stride loop
15:     end while
16:     __syncthreads()
17:     for $s = blockDim.x/2; s > 32; s <<= 1$ do
18:         if $tid < s$ then
19:             Update-Max($tid, tid + s$)
20:         end if
21:         __syncthreads()
22:     end for
```

MATLAB called *ind2sub* [1]. Figure 5.3 shows the mapping for a $2 \times 2 \times 3$ matrix. For example, linear index 7 is mapped to $(1, 0, 1)$. If we call the function as Ind2Sub(7, 2, 2, 3), the return values will be $(\Delta x, \Delta y, \Delta z) = (1, 0, 1)$.

[1]http://www.mathworks.com/help/matlab/ref/ind2sub.html (Last Date Accessed: May 2016)

Algorithm 4, continued

```
    **if** $tid < 32$ **then**
        Update-Max($tid, tid + 32$)
        Update-Max($tid, tid + 16$)
        Update-Max($tid, tid + 8$)
        Update-Max($tid, tid + 4$)
        Update-Max($tid, tid + 2$)
        Update-Max($tid, tid + 1$)
    **end if**
    **if** $tid == 0$ **then**
        $outData[blockIdx.x] \leftarrow smem[0]$
        $outIndex[blockIdx.x] \leftarrow smemIndex[0]$
    **end if**
    **procedure** Update-Max($tA, tB$)
        **if** $smem[tA] < smem[tB]$ **then**
            $smem[tA] \leftarrow smem[tB]$
            $smemIndex[tA] \leftarrow smemIndex[tB]$
        **end if**
    **end procedure**
**end function**
```

Algorithm 5 Ind2Sub($linearIndex, NX, NY, NZ$)

Note: Input is a non-negative integer $linearIndex$, which is a linear index for an element in a volume of size $NX \times NY \times NZ$. NZ is the fastest running index. NX is the slowest running index. Output is equivalent 3D subscripts.

```
1: $\Delta x \leftarrow \lfloor linearIndex/(NY \times NZ) \rfloor$
2: $\Delta y \leftarrow \lfloor (linearIndex \pmod{NY \times NZ})/NZ \rfloor$
3: $\Delta z \leftarrow linearIndex \pmod{NZ}$
4: **return** $(\Delta x, \Delta y, \Delta z)$
```

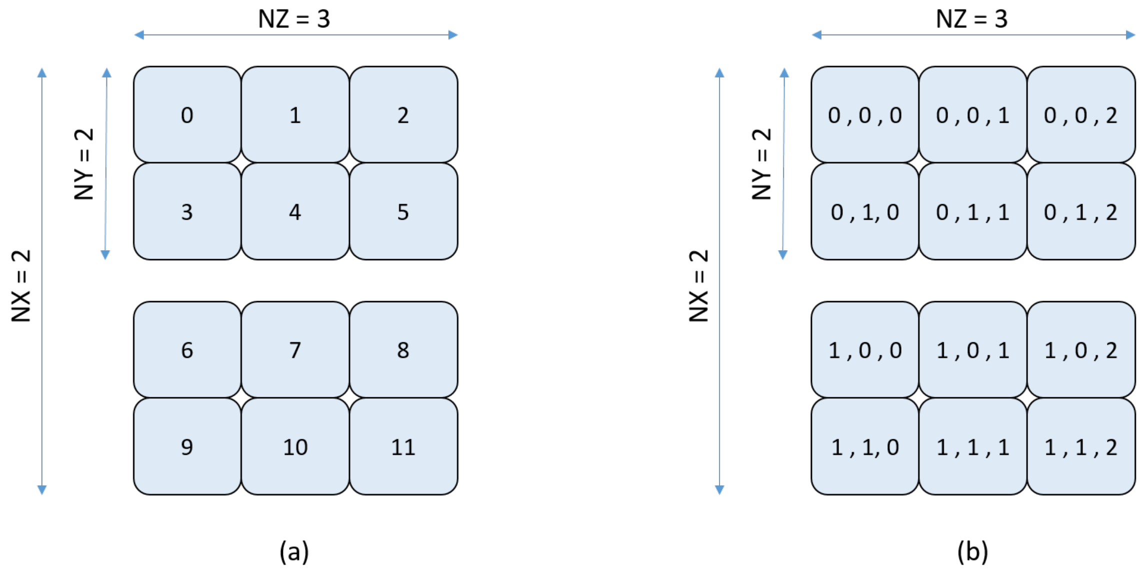


Fig. 5.3.: The mapping from linear indexes in (a) to equivalent subscripts in (b) for a 3D matrix. For (b), each box has the subscripts of (x, y, z).

Algorithm 5 details the implementation. The mapping utilizes modular and floor functions to get the subscript. In C++, assuming $linearIndex, NX, NY, NZ$ are of type integers, we can simply use integer division for the floor function.

6. 2D NORMALIZED CROSS-CORRELATION IN GPU

Although Normalized Cross-Correlation (NCC) is a robust template matching algorithm, its huge amount of computation becomes a challenge. In [23], J.P. Lewis tried to speed up the algorithm by pre-computing integral images over the search window. Based on this approach, MATLAB function *normxcorr2* [1] computes the NCC between two images (a template image and a reference image). However, this function yields unreliable result when the template image has the same size with the reference image. In this case, the computation is incorrect for borders of the coefficient map where the template image does not wholly overlap with the reference image as noted by Dirk Padfield [24]. His implementation of NCC in MATLAB (called *normxcorr2_general* [2]) fixed this problem by normalizing the cross-correlation by the number of actual overlapped pixels. We based on *normxcorr2_general* to implement our GPU version for NCC. This implementation is based on the following formula from [23]:

$$\gamma(u,v) = \frac{\sum_{x,y}[f(x,y)-\bar{f}_{u,v}][t(x-u,y-v)-\bar{t}]}{\left\{\sum_{x,y}[f(x,y)-\bar{f}_{u,v}]^2\sum_{x,y}[t(x-u,y-v)-\bar{t}]^2\right\}^{0.5}} \tag{6.1}$$

where f is the reference image, $\bar{t}$ is the mean of the template image, $\bar{f}_{u,v}$ is the mean of $f(x,y)$ in the region overlapping with t, and γ is the NCC coefficient map.

6.1 Implementation Overview

Algorithm 6 provides an overview of computing our NCC coefficient map. Given a pair of images (say T and A with size (NX,NY) each), function *Normxcorr2* calculates the coefficient map when performing NCC between them. As mentioned in Section 4.2, the size of the coefficient map will be (MX,MY) where $MX = 2NX-1$

[1]http://www.mathworks.com/help/images/ref/normxcorr2.html (Last Date Accessed: May 2016)

[2]http://www.mathworks.com/matlabcentral/fileexchange/29005-generalized-normalized-cross-correlation (Last Date Accessed: May 2016)

and $MY = 2NY - 1$. All variables in the left hand side from Line 2 to Line 16 in Algorithm 6 implicitly have the size of (MX, MY) unless mentioned otherwise.

Lines 1 to 2 compute a matrix of size (MX, MY) which contains the number of overlapped pixels when moving a template image across the entire reference image (shown in Figure 6.1). In this figure, the size of T and A is $(NX, NY) = (2, 2)$, and the size of *overlapPixels* is $(MX, MY) = (2 + 2 - 1, 2 + 2 - 1) = (3, 3)$. To compute *overlapPixels*, function *LocalSum* is utilized with the input being a matrix of all ones of size (NX, NY) named *ones*. The detail of *LocalSum* will be described in Algorithm 8. Creating *ones* requires only a simple GPU kernel that writes 1 to an output array of size $NX \times NY$. This can be done with a grid-stride loop as discussed in Section 5.2.1.

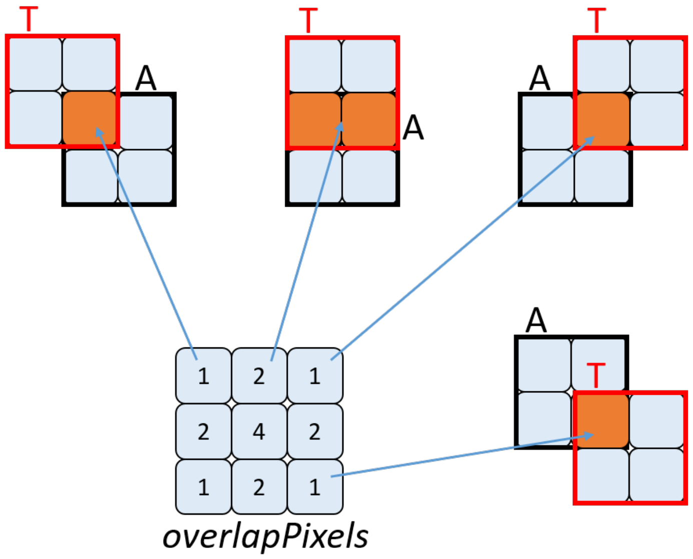


Fig. 6.1.: The number of overlapped pixels in the coefficient map when moving a template image T across the entire reference image A.

Lines 3 to 5 compute the denominator part of Equation 6.1 for the reference image A. This part can be computed efficiently with the integral images (or running sums) of A and A^2 [23]. *LocalSum* is utilized to calculate these integral images. To compute A^2, a simple GPU kernel that squares all pixels in A can be implemented

Algorithm 6 Normxcorr2 $(T, A, requiredOverlapPixels)$

Note: T refers to a template image and A refers to a reference image. We assume that both T and A have the same size (NX, NY). The output is a coefficient map of size (MX, MY) where $MX = 2NX - 1$ and $MY = 2NY - 1$. The value in the coefficient map is set to 0 for the region that has the number of overlapped pixels fewer than $requiredOverlapPixels$.

```
 1: $ones \leftarrow$ Create a matrix of all ones with size $(NX, NY)$
 2: $overlapPixels \leftarrow$ LocalSum $(ones, NX, NY)$
                                              ▷ Denominator part for $A$
 3: $localSumA \leftarrow$ LocalSum $(A, NX, NY)$
 4: $localSumA2 \leftarrow$ LocalSum $(A^2, NX, NY)$
 5: $denomA \leftarrow localSumA2 - (localSumA)^2./overlapPixels$
                                              ▷ Denominator part for $T$
 6: $rotatedT \leftarrow$ FlipMatrix $(T, NX, NY)$
 7: $localSumT \leftarrow$ LocalSum $(rotatedT, NX, NY)$
 8: $localSumT2 \leftarrow$ LocalSum $(rotatedT^2, NX, NY)$
 9: $denomT \leftarrow localSumT2 - (localSumT)^2./overlapPixels$
                                              ▷ Numerator part
10: $fftA \leftarrow \mathcal{F}(A)$
11: $fftT \leftarrow \mathcal{F}(rotatedT)$
12: $xcorrTA \leftarrow \mathcal{F}^{-1}(fftA \circ fftT)$
                                              ▷ Compute the coefficient map
13: $numerator \leftarrow xcorrTA - localSumA \circ localSumT./overlapPixels$
14: $denominator \leftarrow \sqrt{denomA \circ denomT}$
15: $coefMap \leftarrow numerator./denominator$
16: $coefMap\ (overlapPixels < requiredOverlapPixels) \leftarrow 0$
17: return $coefMap$
```

with the same idea of grid-stride loop and one-to-one mapping from an input to an output. Line 5 also requires a simple one-to-one mapping GPU kernel to compute the denominator part for A. We divide $localSumA^2$ element-wise (notated with ./) by $overlapPixels$ to ensure a correct normalization factor for the area where T does not overlap wholly with A. Note that $denomA$ should be non-negative; however, it may contain negative values because of numerical errors [24]. Therefore, whichever values of $denomA$ that are negative will be replaced with zero.

Similarly, the denominator part of T can be computed from Lines 6 to 9. Here, we flip T in both dimensions with function *FlipMatrix* (Algorithm 10) to create *rotatedT* (the reversed template). This is done to match the numerator part of Equation 6.1, which can be done by a convolution between A and $rotatedT$.

Next, Lines 10 to 12 compute the above convolution using Fourier Transform. Here, we ignore the detail of creating and executing plans using cuFFT because it has been explained in Algorithm 1. Although direct convolution in the spatial domain can be used for small sizes of A and T, for large sizes and for the case where the sizes of T and A are the same, the convolution is done more efficiently with multiplication in the frequency domain. Note that we may want to zero-pad T and A to have the sizes of the power of two using Algorithm 9 before computing the Fourier Transform to speed up the computation. After that, $xcorrTA$ needs to be resized back to (MX, MY). When using cuFFT library function for the backward transform, we also need to divide $xcorrTA$ by its size to obtain correct result.

After that, Lines 13 to 15 compute the numerator and denominator of the coefficient map. This step closely follows Equation 6.1. Because all steps are element-wise operation, we combined the computation from Lines 13 to 16 into a single kernel. This is a concept known as kernel fusion [25], which can reduce the data movement from and to the global memory. In addition, common operations such as calculating thread indexes or kernel invocation are done only once. As a result, because there is less redundancy, the execution time will be faster. Note that in Line 16, the value in the coefficient map is set to 0 for the region that has the number of overlapped pixels fewer than *requiredOverlapPixels*. This step is necessary because coefficient values in this region become increasingly unreliable owing to the small number of pixels that overlap. This can result in large coefficient values that approach 1, which would erroneously suggest a perfect match.

6.2 Computing the Local Sum

6.2.1 Row-wise Inclusive Scan

Scan is a popular operation for parallel algorithms. Mathematically, given an array A with N elements such as $[a_0, a_1, \cdots, a_{n-1}]$ and a binary operator $\oplus$, an inclusive scan operation returns the output array as $[a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \cdots \oplus a_{n-1})]$ [26]. For example, when $\oplus$ is the summation operator $(+)$, each element in the output array has the sum of all previous elements including itself.

Our approach of computing local sum follows Hillis and Steele's formulation of parallel scan with a double-buffered version [26]. This is a step-efficient algorithm because the number of steps required to compute the scan for an array of size N is $\mathcal{O}(logN)$ while the number of steps for a sequential implementation is $\mathcal{O}(N)$. An example is shown in Figure 6.2 where the array of size 8 requires only 3 steps for computing a scan.

We also expand the scan operation in our approach as follows. Given an array A with N elements such as $[a_0, a_1, \cdots, a_{n-1}]$ and a binary operator $\oplus$, our scan operation returns the output array of $2N - 1$ elements as $[a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \cdots \oplus a_{n-1}), (a_1 \oplus a_2 \cdots \oplus a_{n-1}), (a_2 \oplus a_3 \cdots \oplus a_{n-1}), \cdots, a_{n-1}]$. This modification is helpful when computing *LocalSum* in Section 6.2.2.

Algorithm 7 describes how to compute the scan for each row of a given matrix. Lines 1 to 4 prepare parameters and launch the MATRIX-ROW-SCAN-KERNEL kernel. The parameters include the number of threads per block, the number of blocks needed, and the size of shared memory. The number of threads per block is chosen to be the maximum block size (1024 threads) in our GPU. The number of blocks are the number

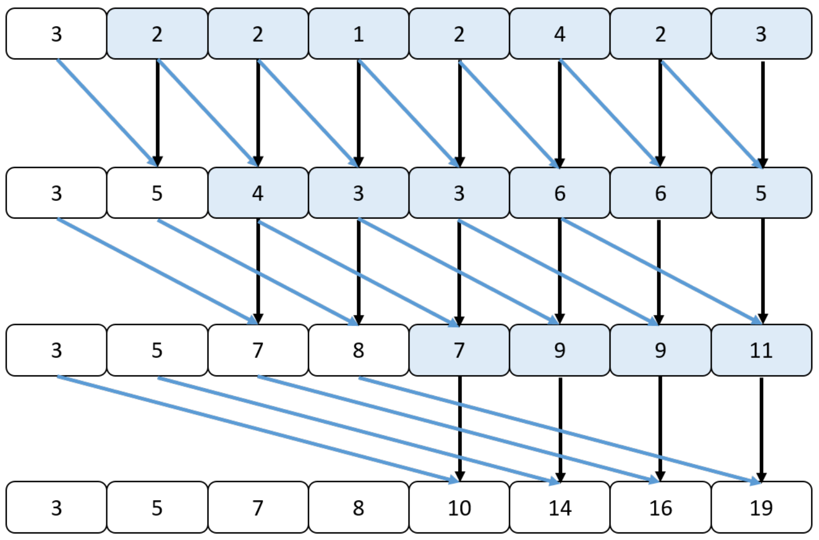


Fig. 6.2.: Hillis and Steele's inclusive scan algorithm.

of rows in the input matrix A. Shared memory is dynamically allocated with size of twice NX with the purpose of double buffering.

In Matrix-Row-Scan-Kernel, Lines 7 to 12 compute thread indexes. Note that we check if a thread index is out-of-boundary in Line 9. Line 13 initializes the pointers to the double buffer in the shared memory *temp*: one for input *pin* and the other for output *pout*. Lines 14 to 15 load data from global memory to shared memory. Function __syncthreads() is needed here to ensure that all threads have finished loading data. Lines 16 to 24 perform the scan for a row. In each iteration, pointers *pin* and *pout* are swapped. After that, only the threads that have indexes larger than $offset$ are active and compute the summation. We also requires __syncthreads() before starting a new iteration. When the for loop is done, we write data from shared memory to output in Line 25.

Typically, a row scan operation stops here. However, we also need to compute remaining elements from index NX to $MX-1$. Here, we take advantage of the result that is already in shared memory and write data to output (Lines 26 to 29).

Algorithm 7 MatrixRowScan (A, NX, NY)

Note: A refers to an input matrix with size (NX, NY). The output (overwritten to A) is a matrix of size (MX, MY) where $MX = 2NX - 1$ and $MY = NY$.

```
1: $threads \leftarrow 1024$
2: $blocks \leftarrow NY$
3: $smemSize \leftarrow 2 \times NX \times sizeof(float)$
4: MATRIX-ROW-SCAN-KERNEL $\lll blocks, threads, smemSize \ggg$ ($A$, $NX$, $NY$, $MX$ )
5: return $A$

6: function MATRIX-ROW-SCAN-KERNEL$(A, NX, NY, MX)$
7:     extern __shared__ float $S[]$
8:     $tx \leftarrow threadIdx.x$
9:     if $tx \geq NX$ then                              $\triangleright$ out-of-boundary
10:        return
11:    end if
12:    $gtid \leftarrow blockIdx.x \times MX + tx$
13:    $pout \leftarrow 0, pin \leftarrow 1$
14:    $S[pout \times NX + tx] \leftarrow A[gtid]$
15:    __syncthreads()
16:    for $offset = 1; offset < NX; offset <<= 1$ do
17:        $pout \leftarrow 1 - pout, pin \leftarrow 1 - pout$
18:        if $tx \geq offset$ then
19:            $S[pout \times NX + tx] \leftarrow S[pin \times NX + tx] + S[pin \times NX + tx - offset]$
20:        else
21:            $S[pout \times NX + tx] = S[pin \times NX + tx]$
22:        end if
23:        __syncthreads()
24:    end for
```

Algorithm 7, continued

```
25:     $A[gtid] \leftarrow S[pout \times NX + tx]$
26:     $endValue \leftarrow S[pout \times NX + NX - 1$
27:     $gtid \leftarrow gtid + NX$
28:     **if** $tx < NX - 1$ **then**
29:         $A[gtid] \leftarrow endValue - S[pout \times NX + tx]$
30:     **end if**
31: **end function**
```

6.2.2 Local Sum

To compute a running sum in 2D, we need to perform *MatrixRowScan* twice. Figure 6.3 shows the data flow for each step of Algorithm 8. First, zero-padding is done for the input matrix of size (NX, NY), making a new matrix of size (MX, MY) where $MX = 2NX - 1$ and $MY = 2NY - 1$. Line 2 will call function *ZeroPadding*, which will call *ZeroPaddingKernel* in Algorithm 9. After that, *MatrixRowScan* is called to perform a scan for each row. Now we need to perform a scan for each column as well. However, to maintain coalesced memory accesses, we need to transpose the matrix and then apply *MatrixRowScan* again. The matrix transpose kernel is adopted from [27]. The result is transposed again to have size (MX, MY).

6.3 Supplementary Kernels

6.3.1 Zero-padding

The zero-padding kernel introduced in this section is for a 3D matrix (a volume) for generality, which means it is also applicable to zero-padding a 2D matrix.

Figure 6.4 demonstrates the zero-padding procedure for a small volume. Let $(AX, AY, AZ) = (3, 2, 2)$ be the size of an input volume. The data layout for this 3D

Algorithm 8 LocalSum (A, NX, NY)

Note: A refers to an input image of size (NX, NY). The output is a matrix of running sum for A with the size of (MX, MY) where $MX = 2NX - 1$ and $MY = 2NY - 1$.

```
1: cudaMemset $(Apad, 0, sizeof(float) \times MX \times MY)$
2: ZeroPadding$(A, NX, NY, 1, Apad, MX, MY, 1)$
3: MatrixRowScan$(Apad, NX, NY)$
4: $Apad \leftarrow Apad^T$
5: MatrixRowScan$(Apad, MY, MX)$
6: $Apad \leftarrow Apad^T$
7: return $Apad$
```

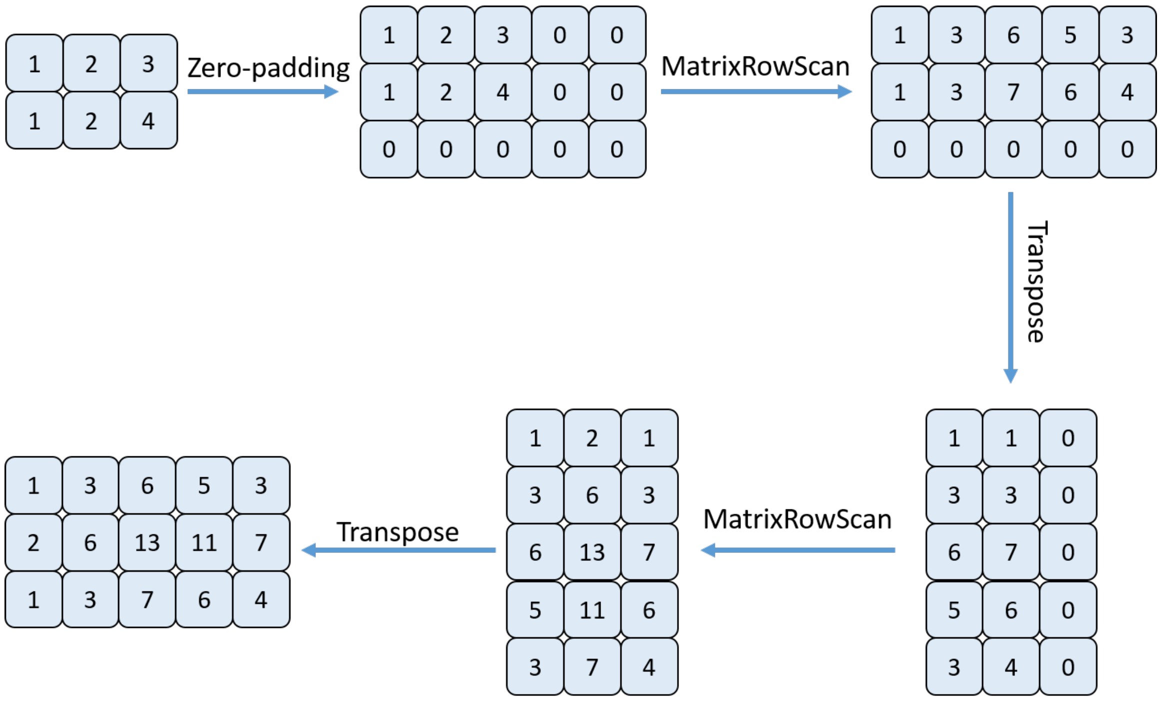


Fig. 6.3.: Data flow for *LocalSum*.

volume array in a GPU is linearly numbered from 1 to 12. We would like to zero-pad this volume to the size of (BX, BY, BZ). In Figure 6.4, this is chosen conveniently as $(5, 3, 3)$. Zero-padding is often used to speed up FFT computation, which is optimized for dimensions in the power of two.

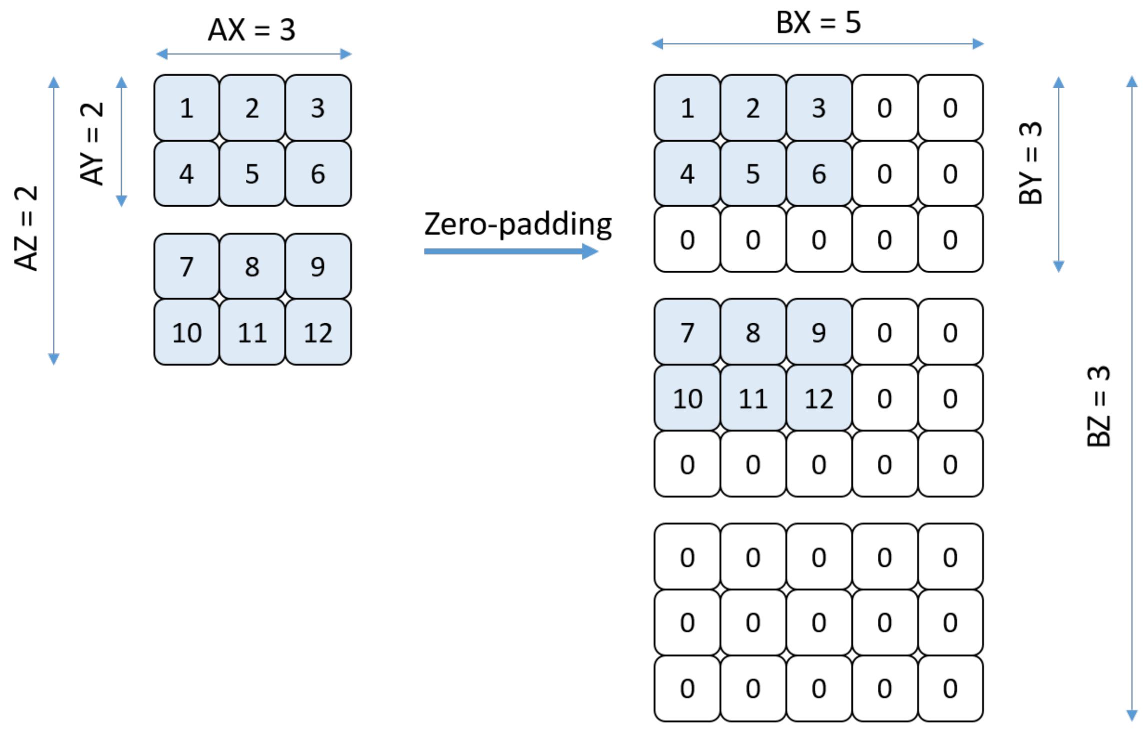


Fig. 6.4.: Zero-padding example for an input volume of size $(3, 2, 2)$.

To prepare for zero-padding, the memory of the padded 3D array must be initialized to an array of $BX \times BY \times BZ$ zeros using the *cudaMemset* function from CUDA SDK. After that, Algorithm 9 copies data from the input 3D array of size (AX, AY, AZ) to the padded 3D array.

Lines 1 to 3 calculate global indexes on each dimension. Line 4 checks if those indexes are within the bounds of (AX, AY, AZ, BX, BY, BZ). Lines 5 and 6 calculate the global memory addresses relative to the sizes of A and B. Finally, Line 7 performs the copy of the elements.

Algorithm 9 ZeroPaddingKernel $(A, AX, AY, AZ, B, BX, BY, BZ)$

Note: A is a pointer to the input volume, B is a pointer to the output volume, (AX, AY, AZ) is the size of A and (BX, BY, BZ) is the size of B

```
1: $tx \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
2: $ty \leftarrow blockIdx.y \times blockDim.y + threadIdx.y$
3: $tz \leftarrow blockIdx.z \times blockDim.z + threadIdx.z$
4: if $tx < AX, BX$ and $ty < AY, BY$ and $tx < AZ, BZ$ then
5:     $gtidA \leftarrow tz \times AX \times AY + ty \times AX + tx$
6:     $gtidB \leftarrow tz \times BX \times BY + ty \times BX + tx$
7:     $B[gtidB] \leftarrow A[gtidA]$
8: end if
```

This kernel can also reverse a zero-padded matrix back to its non-padded version by simply swapping the parameters (A, AX, AY, AZ) and (B, BX, BY, BZ).

6.3.2 Flip a Matrix in 2D

Flipping a matrix in 2D is needed to reverse a template image in Line 6 in Algorithm 6. Figure 6.5 shows an example of flipping a matrix of size 3×2.

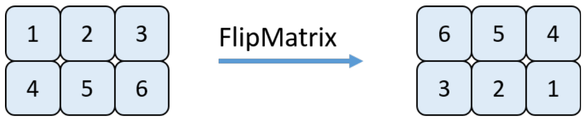


Fig. 6.5.: Example of flipping a matrix in 2D.

Algorithm 10 expanded the example of reversing an array with CUDA in [28]. Lines 1 to 4 prepare parameters to call Flip-Matrix-Kernel. Shared memory is dynamically allocated to have the size of a row in the input matrix. The key point in this kernel is to load data from global memory to shared memory in reverse order (Lines 13 - 14). Having data for each row in A reversed in shared memory allows coalesced memory writing to output B. Note that reversing in the column dimension is done by calculating the output index in Line 15. The block index of the input is $blockIdx.x$ while the block index of the output is $gridDim.x - 1 - blockIdx.x$.

Algorithm 10 FlipMatrix (A, NX, NY)

Note: A refers to an input matrix with size (NX, NY). The output is a matrix of the same size.

```
1: $threads \leftarrow 1024$
2: $blocks \leftarrow NY$
3: $smemSize \leftarrow NX \times sizeof(float)$
                    ▷ B is the output and assumed to be allocated with size $(NX, NY)$
4: FLIP-MATRIX-KERNEL $\lll blocks, threads, smemSize \ggg$ (A, B, NX )
5: **return** $B$

6: **function** FLIP-MATRIX-KERNEL(A, B, NX)
7:     extern __shared__ float $temp$[]
8:     $tx \leftarrow threadIdx.x$
9:     **if** $tx \geq NX$ **then**                              ▷ out-of-boundary
10:        **return**
11:    **end if**
12:    $gtid \leftarrow blockIdx.x \times MX + tx$
13:    $temp[NX - 1 + tx] \leftarrow A[gtid]$
14:    __syncthreads()
15:    $out \leftarrow (gridDim.x - 1 - blockIdx.x) \times NX + tx$
16:    $B[out] \leftarrow temp[tx]$
17: **end function**
```

7. AO-OCT 3D VOLUME REGISTRATION ALGORITHM

7.1 General Steps

Because of eye motion artifacts in AO-OCT volume images, simply aligning a target volume to a reference volume as one piece does not yield a good result. As shown in Figure 2.2, eye motion is more prominent between subsequent fast B-scans (along the y-direction). The reason for this is that the acquisition time for each fast B-scan is so quick that eye motion does not typically occur within one fast B-scan. Therefore, our goal is to align each fast B-scan of a target volume to a correct position in a reference volume by determining shift amounts in 3D as $(\Delta x, \Delta y, \Delta z)$.

However, searching the whole reference volume per fast B-scan incurs a huge computational cost. Therefore, we perform a coarse-to-fine approach to reduce search space. Here we summarize processing steps to register images and later explain each step in detail.

1. Zero padding fast B-scans
2. Coarse registration using sampling and phase correlation
3. Fine registration (Stripe-wise registration) using phase correlation
4. Filter the result to reject incorrect match
5. Visualization of registered result

7.2 Zero-padding Fast B-scans

There are many reasons for zero padding images. First, phase correlation considers an image to be wrapped around, and thus, it is suggested that zero padding or a window function like Hanning window should be applied to eliminate the boundary effect. In our approach, zero padding is done on each fast B-scan (the z and x axes). Second, the target image and the reference image in phase correlation computation need to have the same size. Third, we would like to pad the size of image to the power of two because the FFT algorithm performs best for that size, especially for GPU implementation.

7.3 Coarse Registration using Sampling and Phase Correlation

Next, in our coarse registration approach, instead of finding shift amounts of each fast B-scan, we select some samples in a target volume across the y-axis, where each sample consists of three consecutive fast B-scans. The samples are not adjacent but apart from each other with a uniform distance. We then apply the phase correlation algorithm to each sample to find its correct position in a reference volume image. Using the shift amounts $(\Delta x, \Delta y, \Delta z)$ of the registered samples, we can interpolate to predict the expected $(\Delta x, \Delta y, \Delta z)$ for each fast B-scan of the target volume.

For example, assuming that there are 216 fast B-scans in a target volume, which are indexed from 1 to 216, we can select 8 samples [(1,2,3); (31,32,33); (62,63,64); $\cdots$; (214,215,216)]. If sample (1,2,3) has $\Delta y = 10$ and the sample (31,32,33) has $\Delta y = 42$, we can predict the fast B-scans indexed from 1 to 32 to have linearly increasing Δy from 10 to 42. The phase correlation step is similar to the one described in Section 4.3. However, we perform those steps in a 3D fashion. In addition, phase correlation requires two volumes of the same size. Therefore, in order to phase correlate a sample stripe with a reference volume, additional zero padding on the y-axis is needed for each sample.

We do coarse registration first (with multiple fast B-scans per sample) instead of going directly to fine registration (only use one fast B-scan) because of many reasons. First, this will avoid mis-registration due to the structural similarity of the images (different fast B-scans look very similar). Second, localizing the match position allows us to reduce the search space for each fast B-scan during the fine registration stage. Note that our 3D phase correlation algorithm is very computationally intensive for a large volume size. In our case, it can be $216 \times 318 \times 180$, which results in more than 12 million voxels (3D pixels). Because the correlation-based registration method has the time complexity of $\mathcal{O}(N \log N)$ where N is the total number of voxels, reducing N will gain a more than linear speedup. In summary, overall benefits of coarse registration are higher accuracy and less computation cost for the fine registration step.

7.4 Fine Registration (Stripe-wise Registration)

This step performs registration for individual fast B-scans of a target volume, which would allow us to offset the effect of eye-motion as much as possible. We will denote a fast B-scan that we are referencing as β in the following paragraphs.

With the prediction of shift amounts from the coarse registration step, we can significantly reduce search space for β. Instead of searching the whole reference volume, we can now search in a sub-volume consisting of a few consecutive fast B-scans of the reference volume. However, we need to make sure that this sub-volume indeed contains β.

Let us denote shift amount for β predicted from coarse registration as $(\Delta x_p, \Delta y_p, \Delta z_p)$. Experimentation on various volumes shows that this shift prediction is usually very close (within ± 3 pixels in the y-direction) to the actual shift amount. On a target volume that has extreme eye motion, error in prediction can go up to ± 10 pixels in the y-direction. Therefore, to make sure that the chosen sub-volume contains β, we select a sub-volume in the reference image that consists of fast B-scans that range in $[\Delta y_p - w_y, \Delta y_p + w_y]$ where w_y is a window size typically chosen as 16. Note that we only limit the boundary of the sub-volume on y-direction but not x and z directions because of two reasons. First, prediction error on x and z directions varies widely. Second, memory layout of 3D matrix in the GPU allows easier slicing on the y-direction.

After that, we perform phase correlation between β and the selected sub-volume to find $(\Delta x, \Delta y, \Delta z)$ in the same manner as the coarse registration step. However, because this Δy is the shift amount in relative to the sub-volume rather than the reference volume, we must add to Δy the corresponding offset $(\Delta y_p - w_y)$.

7.5 Filter the Results to Reject Incorrect Match

After the fine registration step, there may appear incorrect registration results for some fast B-scans. This usually happens for those fast B-scans in a target volume which do not overlap on any of those in the reference volume. We can detect these fast B-scans by comparing their shift amounts with the B-scans in the neighborhood. If the difference in the shift amount is too extreme, it is assumed to be an incorrect match. After that, we can exclude those out-of-bound fast B-scans when reconstructing a registered volume.

7.6 Visualization of the Registered Result

Now, we can shift each fast B-scan in a target volume based on the calculated $(\Delta x, \Delta y, \Delta z)$ and reconstruct a registered volume. In addition, projection images (slow B-scan and C-scan) can be made to visualize the effect of registration. Figure 7.1 demonstrates the registration of one stripe of fast B-scans in a target volume. After performing registration, the match position is found at $(\Delta x, \Delta y, \Delta z) = (11, 49, 70)$. By registering all fast B-scans in a target volume, we can reconstruct a registered volume with the projections shown in Figure 7.1 (c) and (f). The black gaps appear in the y-direction because eye motion in that direction was not linear during acquisition of the volume. Also, when we reconstruct a volume by moving each fast B-scan to its $(\Delta x, \Delta y, \Delta z)$ location, it may cause overlapping.

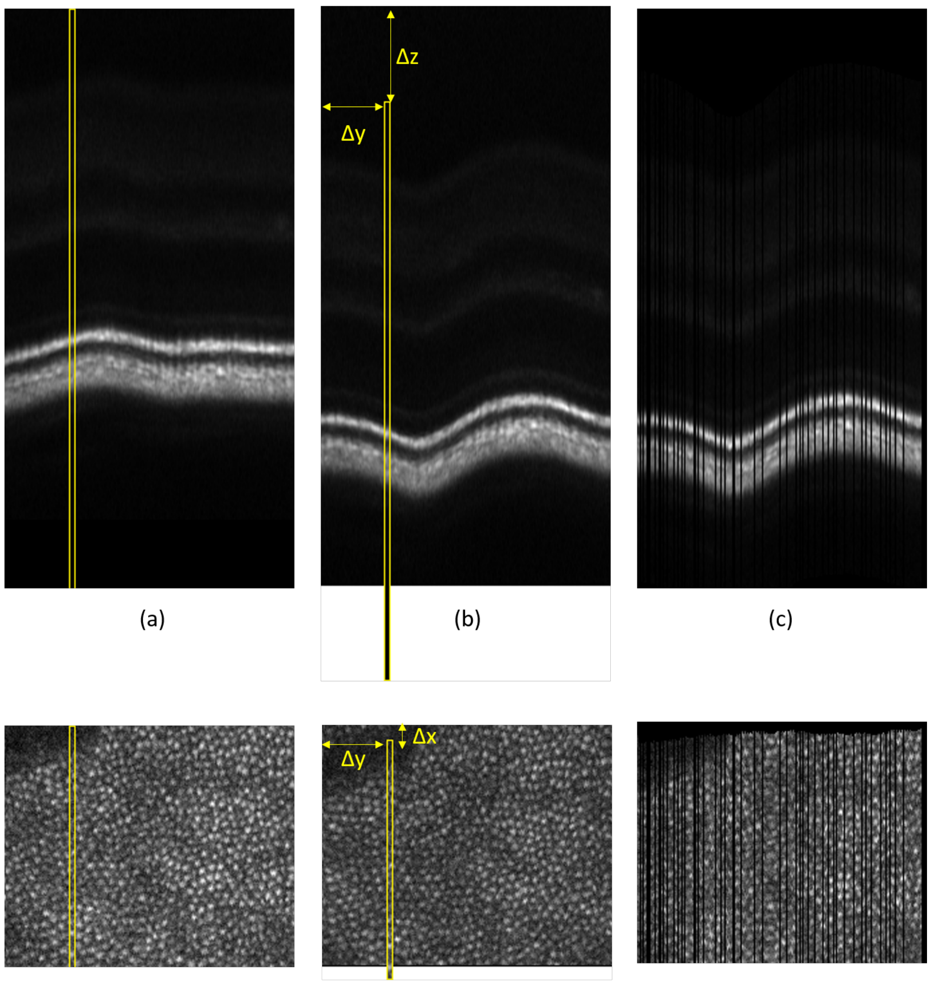


Fig. 7.1.: Registration of one stripe of fast B-scans in a target volume in 3D. To visualize registration, we show projection images here. (a) Slow B-scan projection of a target volume, with the yellow stripe to be registered. (b) Slow B-scan projection of a reference volume, with the registered location of the selected stripe. (c) Slow B-scan projection of the entire registered volume. (d) C-scan projection of the original target volume with the yellow stripe to be registered. (e) C-scan projection of the reference volume with the yellow registered stripe. (f) C-scan projection of the entire registered volume.

8. IMPLEMENTATION IN GPU

8.1 Implementation Overview

Among a series of 3D volumes captured from the same retina area, one volume with the highest quality is chosen to be a reference volume. This can be done either manually or automatically by finding the volume with the greatest image clarity and minimal eye motion artifacts. The task then is to register each target volume by aligning its fast B-scans to the reference volume.

The implementation takes advantage of both CPU and GPU depending on the type of computations required. For example, for tasks with large data size like FFT, GPU is utilized while for tasks with small data size like interpolation, CPU is utilized. In addition, we perform caching of duplicate computations to avoid redundant computation. Specifically, phase correlation requires two forward FFTs (one for a target volume and the other for a reference volume) and one backward FFT (for the phase correlation). However, we can just compute the forward FFT for a reference volume once and keep reusing the result.

The overall algorithm is summarized in Algorithm 11.

8.2 Memory Allocation and Pre-computation

Because the data initially resides on the CPU side, we need to copy all of the volumes to the GPU. The data of double precision will be converted to single precision floating point numbers to reduce the data size by half. In this pre-computation step, we first copy the reference volume and apply zero-padding for each fast B-scan. The effect of zero-padding is to avoid the boundary effect of phase correlation. In addition, FFT algorithm in GPU works best for the size of power of two. For zero-padding, we

Algorithm 11 3D Registration

Note: Stripe width refers to the number of fast B-scans chosen as a target volume to be used in a group for phase correlation. In coarse registration, the stripe width is 3, while in fine registration, the stripe width is 1.

```
1:  Memory allocation and pre-computation
2:  for each target volume do
                                                        ▷ Coarse Registration
3:      for each sample of stripe width 3 do
4:          $(\Delta x_p, \Delta y_p, \Delta z_p) \leftarrow$ PhaseCorrelation (sample, reference volume)
5:      end for
6:      Shift prediction
                                                          ▷ Fine Registration
7:      for each fast B-scan $\beta$ in the target volume do
8:          Determine reference sub-volume based on $(\Delta x_p, \Delta y_p, \Delta z_p)$
9:          $(\Delta x, \Delta y, \Delta z) \leftarrow$ PhaseCorrelation ($\beta$, sub-volume)
10:         Offsetting $\Delta y$ based on sub-volume index
11:     end for
12:     Filter the registration results
13: end for
```

utilized Algorithm 9 with the following input parameters: AX is the A-line length, AY is the number of A-lines per fast B-scan, and AZ is the number of fast B-scans per volume. The output parameters BX and BY will be chosen as a power of two. However, BZ is chosen as the closest multiple of 32 to avoid requiring too much memory.

Knowing the size of the reference volume, we can allocate required memories for the following items:

- The reference volume and its DFT
- A series of sub-volumes in the reference volume and their DFTs (to be introduced later)
- A target volume and its DFT
- The plans for cuFFT for forward transformations and backward transformations
- Outputs for the FIND-MAX-INDEX-KERNEL KERNEL

After this, each target volume will be copied from the host to the GPU to the allocated memory sequentially. The coarse registration step begins in the next section.

8.3 Coarse Registration

To choose fast B-scan indexes for the samples (as mentioned in Section 7.3) in a target volume, we can simply create a linearly-spaced array as shown in Algorithm 12. This is similar to the function *linspace* from MATLAB [1]. For example, assuming that our target volume has 216 fast B-scans, we can call LinearSpace$(0, 213, 10)$ to create 10 sample indexes for stripes of width 3. The result will then be $samplingIndexes = [0, 24, 47, 71, 95, 118, 142, 166, 189, 213]$. Each sample will consist of three consecutive fast B-scans. For example, the sample at index 213 will consist of fast B-scans of indexes $[213, 214, 215]$.

[1]http://www.mathworks.com/help/matlab/ref/linspace.html (Last Date Accessed: May 2016)

For each sample, we apply the PhaseCorrelation algorithm as mentioned in Section 5.1. Note that because PhaseCorrelation requires that a target volume and a reference volume have the same size, zero-padding is also needed for each selected sample. Then, shifted amount Δy_p will be recorded as $sampledValues$. Figure 8.1 shows $sampledValues$ as solid dots. We expect the dots to follow an upward linear trend because of the order of the acquired B-scans. However, some dots fall out of this trend if they are out-of-bound. If this happens, the registration algorithm would not find a good match position. This is visible for the leftmost dot (of $bScanIdx = 0$) in Figure 8.1.

Algorithm 12 LinearSpace $(start, end, numSamples)$

```
1: $interval \leftarrow (end - start)/(numSamples - 1)$
2: for $i = 0; i < numSamples; i++$ do
3:     $samplingIndexes[i] = round(start + interval \times i)$
4: end for
5: return $samplingIndexes$
```

Therefore, we need to detect the dots that are not in the trend. Algorithm 13 demonstrates this process. Similar to a median filter, this algorithm detects abnormal points based on the distances between those points and their neighbors in the first step (Lines 2 - 8). A point is good if distances (dL and dR) are positive (because we are expecting an upward linear trend) and smaller by a specified *tolerance* (near neighbors). Typically, the *tolerance* is chosen to be twice the distance/interval between the sampled indexes. In this first step, however, some points around a spike may be marked as not good when requiring both dL and dR to be within a threshold (Line 5). Thus, the second step finds those good points that are accidentally marked as not good in the first step. We can do this by relaxing the requirement in Line 12. After this step is done, the function finally returns all the points that are detected as spikes.

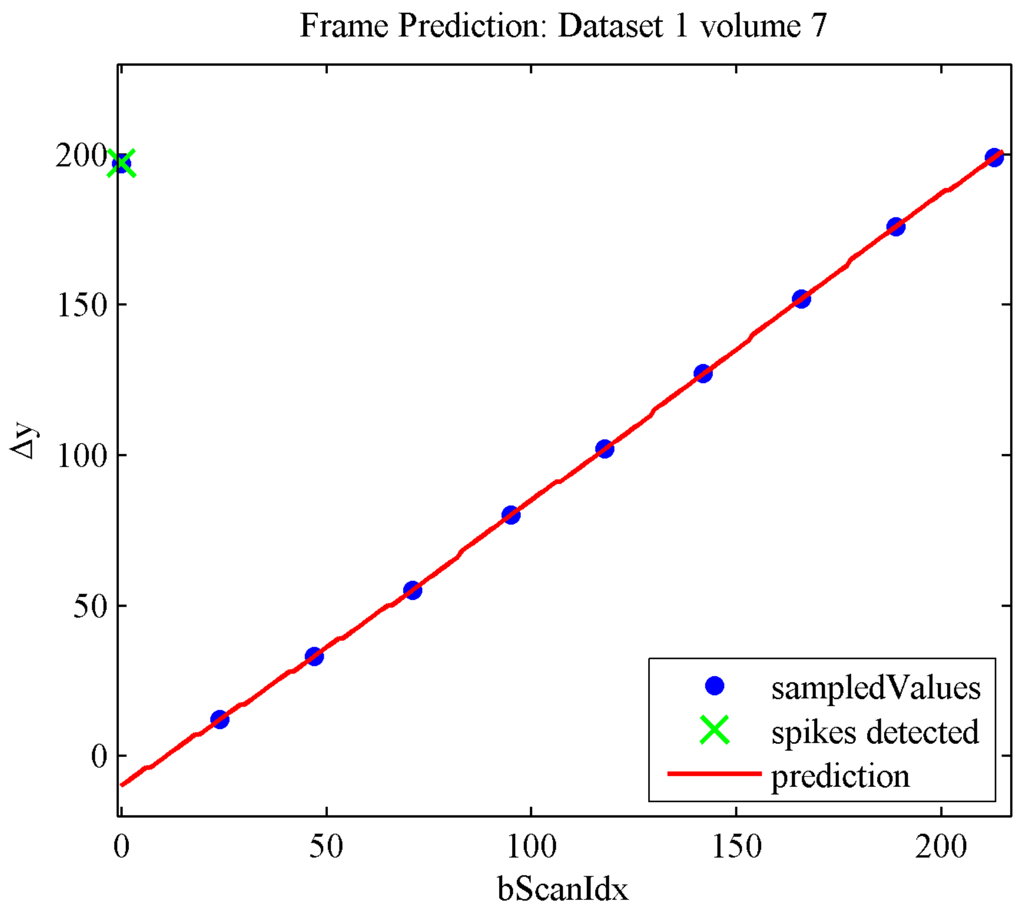


Fig. 8.1.: Frame prediction based on sampled B-Scans.

For example, in Figure 8.1, after running the first step of SpikeDetection on *sampledValues*, the first and second data points are detected as spikes. By applying the second step in SpikeDetection, the second data point will be recovered, specifically, marked as a good point. As a result, the only point that is marked as not good is the first point.

Now, the remaining points that are not marked as spikes are referred to as good points. For those good points, we can interpolate to complete the shift prediction for each fast B-scan across the target volume in the next fine registration stage. This is essential because we can limit the search space significantly, thereby reducing computation time. For a volume of size $512 \times 512 \times 512$, coarse-to-fine approach is about $5000\times$ faster than going directly to the fine registration step.

Algorithm 13 SpikeDetection $(A, tolerance)$

Note: A is the array that we need to detect the spikes. $tolerance$ is the threshold to decide if a point is good or bad.

```
1: Mark all points in A as not good.
2: for $i = 1; i < input.length - 2; i++$ do                          ▷ First step
3:     $dL \leftarrow$ distance between A[i] and A[i-1]
4:     $dR \leftarrow$ distance between A[i+1] and A[i]
5:     if $0 < dL < tolerance$ and $0 < dR < tolerance$ then
6:         Mark A[i] as good.
7:     end if
8: end for
9: for each A[i] that is marked as not good do                        ▷ Second step
10:    $dL \leftarrow$ distance between A[i] and A[i-1]
11:    $dR \leftarrow$ distance between A[i+1] and A[i]
12:    if $0 < dL < tolerance$ or $0 < dR < tolerance$ then
13:        Mark A[i] as good.
14:    end if
15: end for
16: return indexes of A that is not good.
```

8.4 Fine Registration

This step performs a 3D stripe-wise registration similar to the coarse registration step. However, there are two changes. First, we register each individual fast B-scan (denoted as β) in a target volume instead of a group of three. Second, we reduce search space to a sub-volume in the original reference volume rather than using the whole volume. In this way, the entire volume is partitioned into overlapping sub-volumes that, for example, are 32 frames wide and spaced every 16 frame. A frame consists of one fast B-scan.

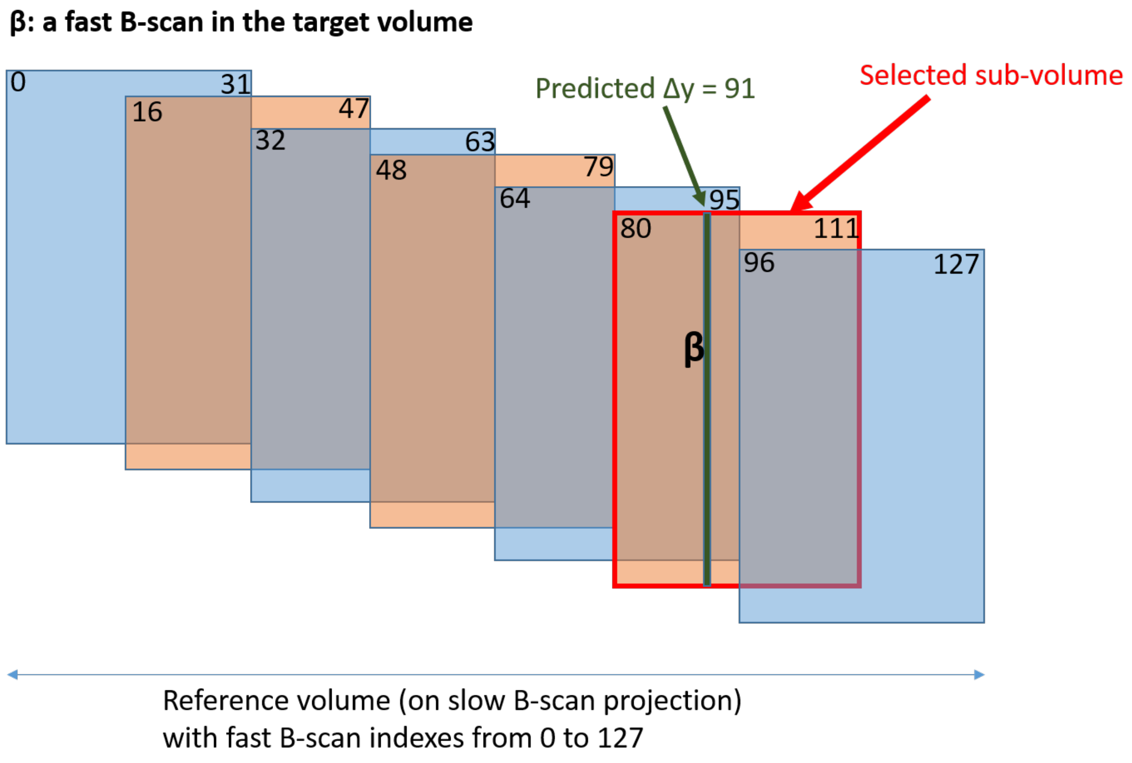

β: a fast B-scan in the target volume
0
31
16
47
32
63
48
79
64
95
80
111
96
127
Predicted Δy = 91
Selected sub-volume
β
Reference volume (on slow B-scan projection)
with fast B-scan indexes from 0 to 127

Fig. 8.2.: Fine registration using a sub-volume from the reference volume.

A sequence of many sub-volumes is apart from each other by a distance of 16 frame interval. Figure 8.2 shows an example when the original reference volume has 128 frames with fast B-scan indexes from 0 to 127. Then, we have seven sub-volumes with fast B-scan indexes as follows: $(0-31), (16-47), (32-63), (48-79), (64-95), (80-111)$, and $(96-127)$. In the pre-computation step in Section 8.2, we can compute the FFTs of these sub-volumes, thus avoiding duplicate computations during PhaseCorrelation. Note that these sub-volume are chosen to be overlapped to one another because of the following reason. For each fast B-scan β in the target volume, we can choose a reference sub-volume such that the predicted position is as near the center of chosen sub-volume as possible (Figure 8.2). Algorithm 14 shows how to select a sub-volume based on Δy_p.

Algorithm 14 DetermineSubVolume ($bScanIdx, totalFrames, subVolSize$)

Note: $bScanIdx$ is the B-scan index that we would like to find the correct sub-volume index. $totalFrames$ is the total number of fast B-scans in the reference volume, $subVolSize$ is the size of the sub-volume.

```
1: $quarterSub \leftarrow subVolSize/4$
2: $numQuarters \leftarrow \lceil totalFrames/quaterSubs \rceil$
3: $numSubVols \leftarrow \lfloor (totalFrames - 1)/(subVolSize/2) \rfloor$
4: $quarterIdx \leftarrow bScanIdx/quarterSub$
5: if $quarterIdx < 3$ then
6:     return 0                                        ▷ Return the first sub-volume
7: else
8:     if $quarterIdx >= numQuarters - 3$ then
9:         return $numSubVols - 1$                     ▷ Return the last sub-volume
10:    else
11:        return $\lfloor (quarterIdx - 3)/2 \rfloor + 1$
12:    end if
13: end if
```

Now we can execute Line 9 in Algorithm 11 to find $(\Delta x, \Delta y, \Delta z)$ by phase correlation between β and the selected sub-volume. Note that Δy calculated here is relative to this sub-volume rather than to the original reference volume. Therefore, we must compensate by adding a corresponding offset to Δy. After this, the shifted amount is recorded for each fast B-scan (shown as x markers in Figure 8.3).

8.5 Filter the Results

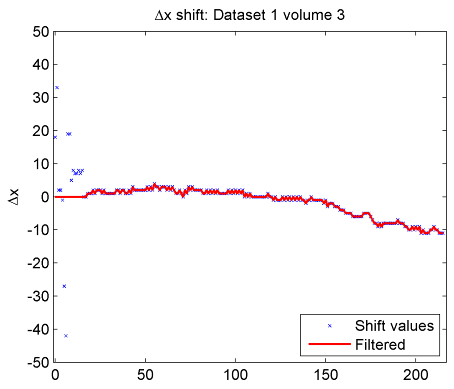
However, there are some spike noises either because of out-of-bound fast B-scans or incorrect registration from PhaseCorrelation. The latter happens when the maximum peak is found at the wrong place. Therefore, we can filter the result to detect these spikes using a modified version of SpikeDetection in Algorithm 13. In this modified version, the *input* in Line 1 is an array of 3D points, each specifying shifted amount $(\Delta x, \Delta y, \Delta z)$ for a fast B-scan. The distance between two points a and b can be Euclidean distance $\|a - b\|$. Because we expect eye movement to be smooth, the *tolerance* value is empirically chosen to be equal to 8 (pixels). A small tolerance will overly detect spike noise, while a high tolerance will miss detecting noise. As we reject the points that are detected as noise, missing points will be filled in by interpolating from remaining values. The missing points at the left and right boundaries are also calculated by extrapolation. The detail is shown in Algorithm 15.

Figure 8.3 shows filtered results as red lines. Note that out-of-bound B-scans from indexes 0 to 19 are fixed.

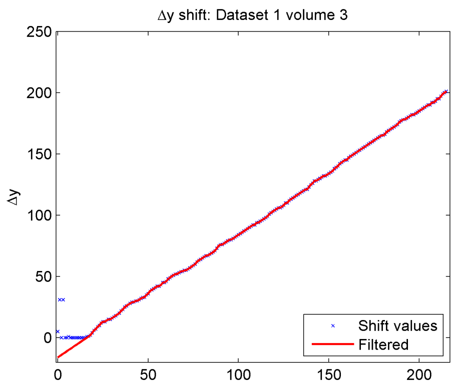
Algorithm 15 ShiftCorrection $(x, y, z, tolerance)$

Note: x, y, z are the three arrays from $(\Delta x, \Delta y, \Delta z)$, $tolerance$ is the parameter to be passed to *SpikeDetection* in Algorithm 13.

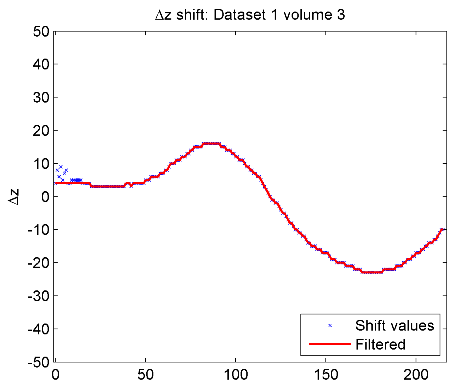
```
1: $badPoints \leftarrow$ SpikeDetection$((x, y, z), tolerance)$
                                        ▷ Extrapolation at the left boundary
2: $yStart \leftarrow$ Find the first data point in $y$ that has value 0.
3: $x[0..yStart] \leftarrow x[yStart]$
4: $y[0..yStart] \leftarrow [-yStart + 1, -yStart + 2, \cdots]$
5: $z[0..yStart] \leftarrow z[yStart]$
                                        ▷ Extrapolation at the right boundary
6: $yEnd \leftarrow$ Find the first data point in $y$ that has value equal to $y.length$
7: $x[yEnd..end] \leftarrow x[yEnd]$
8: $y[yEnd..end] \leftarrow [y[yEnd], y[End] + 1, \cdots]$
9: $z[yEnd..end) \leftarrow z[yEnd]$

10: $(xf, yf, zf) \leftarrow$ Interpolate $badPoints$ using remaining points in (x,y,z).
11: **return** $(xf, yf, zf)$
```



(a) Δx shift



(b) Δy shift



(c) Δz shift.

Fig. 8.3.: Shift amount before and after filtering.

9. EXPERIMENTAL RESULTS

9.1 Description

We have tested our algorithm on various data sets. The results shown here are for a sequence of 12 volumes of size $432 \times 180 \times 216$ each. To visualize the effect of our proposed 3D registration, some projection images including C-scans and slow B-scans are created. In Figure 9.1 and Figure 9.2, the first row shows projection images before registration and the second row shows projection images after registration. Out-of-bound fast B-scans are also excluded. Vol 10 is chosen as the reference volume, to which other volumes (Vol 3, Vol 6, and Vol 11) are registered.

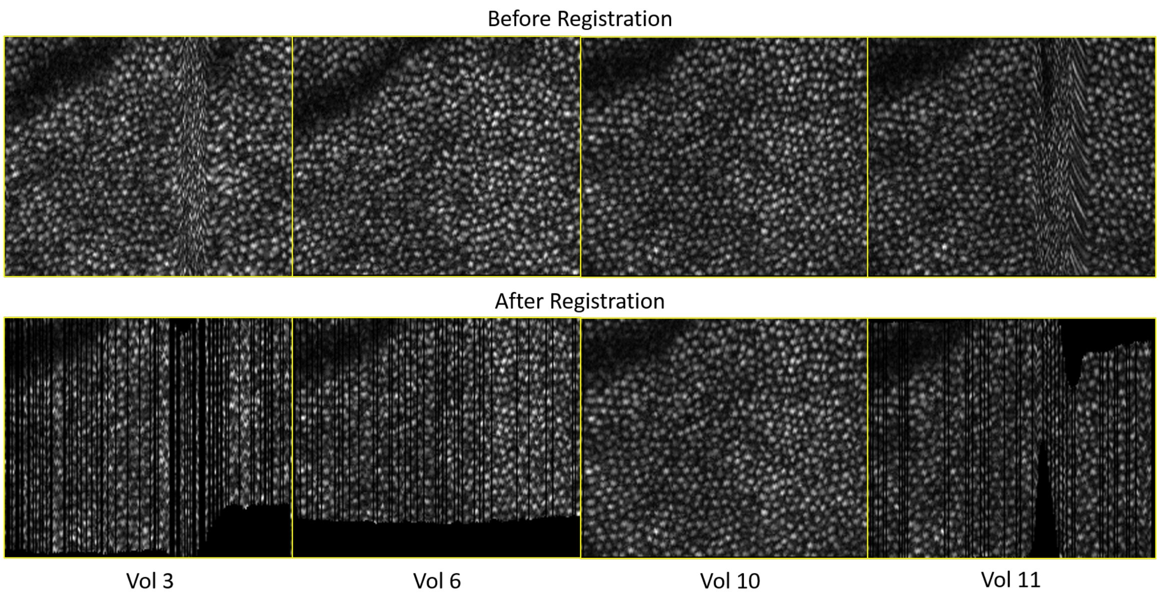


Fig. 9.1.: Registration results visualized on C-scans. Volumes were registered to Vol 10.

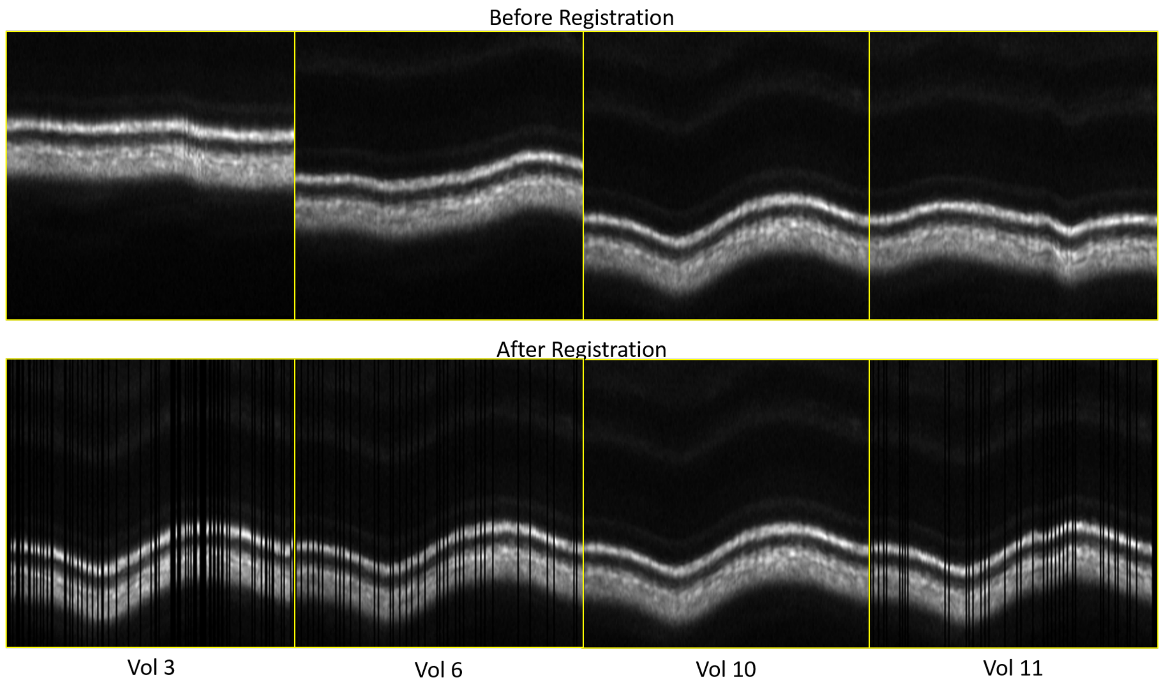
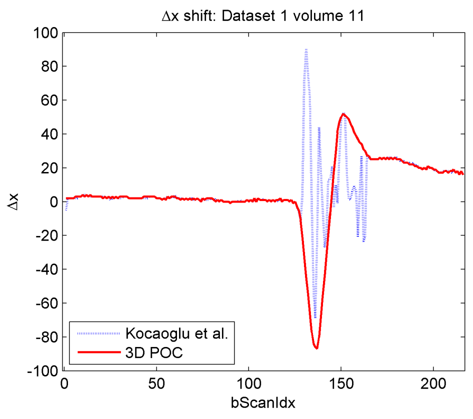
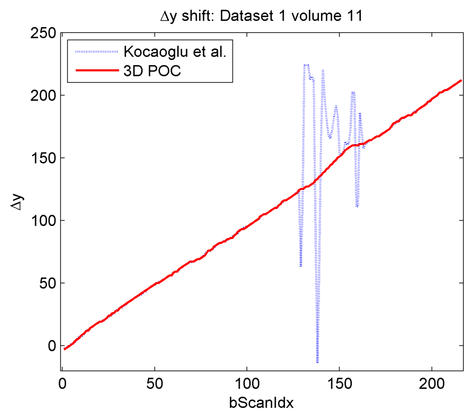


Fig. 9.2.: Registration results visualized on slow B-scans. Volumes were registered to Vol 10.

Note that in this data set, Vol 3 and Vol 11 contain eye motion artifacts, which are visible on their C-scans. In a previous stripe-wise registration method with normalized cross-correlation using only C-scans by Kocaoglu el al. [1, 29], noisy C-scans during micro-saccadic eye movements do not provide reliable registration. However, as our 3D POC utilizes all information from each fast B-scan rather than from the projection image, those B-scans within an eye motion area can be tracked and registered. Figure 9.3 shows the effectiveness of our 3D approach when capturing eye motion appeared in Vol 11. For areas without eye motion, the differences in shift amounts calculated between 3D POC and Kocaoglu's method are within ± 1 pixel for Δx and Δy. Another advantage of 3D POC is that the whole volume is registered without the need of creating projection images or processing extra steps like done in [10]. The Kocaoglu's method consists of two separate steps, namely axial registration and lateral registration, which use projection images such as slow B-scan and C-scan, respectively.



(a) Δx shift



(b) Δy shift

Fig. 9.3.: Comparison between 3D POC with previous method by [10].

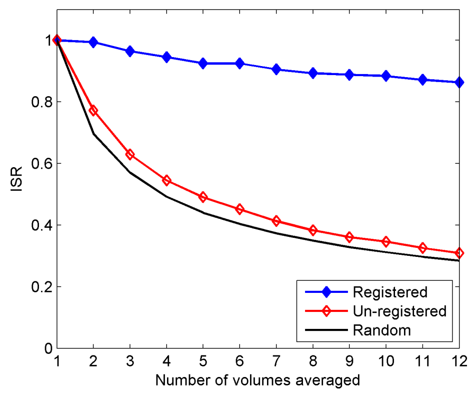
9.2 Registration Accuracy

To quantify the effectiveness of image registration, we create a sequence of averaged C-scans similar to the method described in [30]. Note that for easy visualization, averaged C-scans are used instead of averaged volumes. To compare an averaged C-scans to a reference C-scan, the following two metrics are used: Image Sharpness Ratio (ISR) and Structural Similarity Index (SSIM). Image sharpness for an image (say I) is estimated from image gradients calculated by $\|\nabla I\|$ [1]. ISR is the ratio of image sharpness between an averaged C-scan to a chosen reference C-scan. The other metric SSIM is used to measure image quality based on three characteristics including luminance, contrast, and structure [2]. Figures 9.4(a) and 9.4(b) plot these metrics as a function of the number of volumes used for averaging. As a baseline, we also show ISR and SSIM in the case of a simulated image sequence created by *rand*

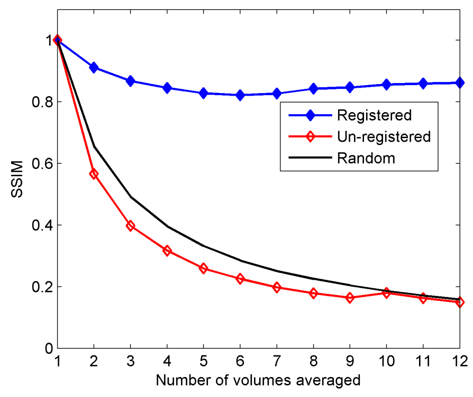
[1]http://www.mathworks.com/matlabcentral/fileexchange/32397-sharpness-estimation-from-image-gradients (Last Date Accessed: June 2016)

[2]http://www.mathworks.com/help/images/ref/ssim.html (Last Date Accessed: June 2016)

function in MATLAB, which outputs uniformly distributed random numbers. With registration, the averaged image maintains much higher ISR and SSIM values compared to without registration. This difference is visually evident in the corresponding averaged C-scans in Figure 9.5.

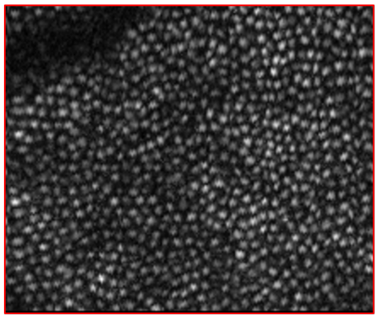


(a) ISR

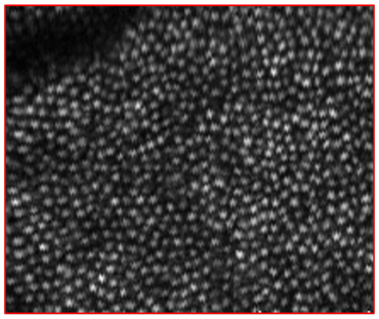


(b) SSIM

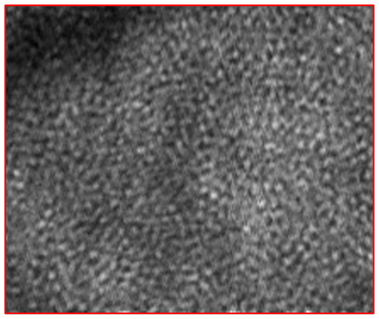
Fig. 9.4.: Metrics to quantify image registration.



(a) Reference C-scan



(b) Averaged C-scan with registration



(c) Averaged C-scan without registration

Fig. 9.5.: Averaged C-scan with and without registration compared with the reference C-scan.

9.3 Registration Speed

One drawback of using 3D POC is that the computation time is slower as compared to the Kocaoglu's method, which was done with 2D projection images (3D POC is about 3× slower). However, with GPU implementation, the time is reduced significantly. To measure speedup, we compare the execution time to register one target volume to a reference volume between two implementations of 3D POC. The first version is a serial program in MATLAB while the second is a parallel program in CUDA/C++. The experiment is performed on a computer with an *Intel*® Core i7 5930K CPU @ 3.50 GHz, 32 GB RAM, and a Titan X GPU. The machine runs a 64-bit Windows operating system. Table 9.1 records the execution time for various volume sizes. For the GPU implementation, the time includes time to copy a target volume from the host to the GPU, perform the computation, and copy the result back to the host.

Table 9.1: Execution time in seconds for CPU version (in MATLAB) and GPU version (in CUDA) with the corresponding speedup. The time shown includes the mean and standard deviation from 10 test runs

Size	MATLAB (CPU)	CUDA (GPU)	Speedup
$64 \times 64 \times 64$	0.73 ± 0.17	0.04 ± 0.00	17.39×
$128 \times 128 \times 128$	3.25 ± 0.20	0.17 ± 0.02	19.32×
$256 \times 256 \times 256$	21.63 ± 0.55	0.80 ± 0.08	27.16×
$512 \times 256 \times 256$	43.22 ± 0.65	1.46 ± 0.14	29.65×
$256 \times 512 \times 256$	43.89 ± 0.92	1.54 ± 0.19	28.56×
$256 \times 256 \times 512$	43.65 ± 0.62	1.60 ± 0.20	27.22×
$512 \times 512 \times 512$	199.30 ± 2.59	5.93 ± 0.66	33.62×

With the pre-computation (caching) technique, a reference volume only needs to be loaded and processed once at the beginning of the algorithm. Thus, the total execution time, which includes the time from reading data from hard disk to the time when all volumes are registered, is reduced. This result demonstrates that large speedups are obtained from our new CUDA implementation, massively parallel GPU hardware, and the implementation of caching in the algorithm.

10. SUMMARY

10.1 Future Work

As the registration method described in this thesis is for a pixel-level shift, an improvement of the algorithm is to use an interpolation technique to align the target volume with a sub-pixel resolution. This will allow a higher resolution image when combining data from various volumes. In addition, other techniques to decrease registration time for GPU can be explored such as overlapping data transfer and kernel execution with streams. Furthermore, the expansion of the implementation for multiple GPUs could be considered. The faster the registration time is, the more suitable it is for on-the-fly real-time applications.

10.2 Conclusion

The thesis describes a framework for a stripe-wise 3D registration method with phase correlation for 3D AO-OCT data. With various techniques such as pre-computation, coarse to fine registration, and GPU implementation, the algorithm achieves not only high registration accuracy but also significant speedup compared to CPU implementation. The thesis also described a spike noise detection algorithm, which was used in both Δy prediction and final result filtering. Using our 3D registration method, we can even track fast eye motions such as micro-saccades, which is an important improvement over a previous stripe-wise registration method with 2D projection images. GPU implementations for two popular correlation-based image registration methods, POC and NCC, are also included in detail.

LIST OF REFERENCES

LIST OF REFERENCES

[1] O. P. Kocaoglu, T. L. Turner, Z. Liu, and D. T. Miller, "Adaptive optics optical coherence tomography at 1 MHz," *Biomedical optics express*, vol. 5, no. 12, pp. 4186–4200, 2014.

[2] I. I. Bussel, G. Wollstein, and J. S. Schuman, "OCT for glaucoma diagnosis, screening and detection of glaucoma progression," *British Journal of Ophthalmology*, pp. bjophthalmol–2013, 2013.

[3] C. V. Regatieri, L. Branchini, and J. S. Duker, "The role of spectral-domain OCT in the diagnosis and management of neovascular age-related macular degeneration," *Ophthalmic Surgery, Lasers and Imaging Retina*, vol. 42, no. 4, pp. S56–S66, 2011.

[4] M. L. Gabriele, G. Wollstein, H. Ishikawa, J. Xu, J. Kim, L. Kagemann, L. S. Folio, and J. S. Schuman, "Three dimensional optical coherence tomography imaging: advantages and advances," *Progress in retinal and eye research*, vol. 29, no. 6, pp. 556–579, 2010.

[5] S. Martinez-Conde, S. L. Macknik, and D. H. Hubel, "The role of fixational eye movements in visual perception," *Nature Reviews Neuroscience*, vol. 5, no. 3, pp. 229–240, 2004.

[6] D. Huang, E. A. Swanson, C. P. Lin, J. S. Schuman, W. G. Stinson, W. Chang, M. R. Hee, T. Flotte, K. Gregory, C. A. Puliafito, *et al.*, "Optical coherence tomography," *Science*, vol. 254, no. 5035, pp. 1178–1181, 1991.

[7] J. S. Schuman, "Spectral domain optical coherence tomography for glaucoma (an AOS thesis)," *Transactions of the American Ophthalmological Society*, vol. 106, p. 426, 2008.

[8] P. A. Calabresi, L. J. Balcer, and E. M. Frohman, *Optical Coherence Tomography in Neurologic Diseases*. Cambridge University Press, 2015.

[9] D. Miller, O. Kocaoglu, Q. Wang, and S. Lee, "Adaptive optics and the eye (super resolution OCT)," *Eye*, vol. 25, no. 3, pp. 321–330, 2011.

[10] O. P. Kocaoglu, S. Lee, R. S. Jonnal, Q. Wang, A. E. Herde, J. C. Derby, W. Gao, and D. T. Miller, "Imaging cone photoreceptors in three dimensions and in time using ultrahigh resolution optical coherence tomography with adaptive optics," *Biomedical optics express*, vol. 2, no. 4, pp. 748–763, 2011.

[11] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.

[12] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[13] "Maxwell: The most advanced CUDA GPU ever made." Internet Source: https://devblogs.nvidia.com/parallelforall, 2014. Last Date Accessed: May 2016.

[14] "GeForce GTX Titan X specifications." Internet Source: www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications, 2015. Last Date Accessed: April 2016.

[15] H. Jeong, S. Lee, W. Lee, J. Pak, J. Kim, and J. Chung, "Performance of GTX Titan X GPUs and code optimization," *33rd International Symposium on Lattice Field Theory (Lattice 2015)*, 2015.

[16] J. N. Sarvaiya, S. Patnaik, and S. Bombaywala, "Image registration by template matching using normalized cross-correlation," in *Advances in Computing, Control, & Telecommunication Technologies, 2009. ACT'09. International Conference on*, pp. 819–822, IEEE, 2009.

[17] R. Gonzalez, "Improving phase correlation for image registration," *Proceedings of Image and Vision Computing New Zealand 2011*, 2011.

[18] M. Balci and H. Foroosh, "Subpixel registration directly from the phase difference," *EURASIP Journal on Applied Signal Processing*, vol. 2006, pp. 231–231, 2006.

[19] H. Foroosh, J. B. Zerubia, and M. Berthod, "Extension of phase correlation to subpixel registration," *IEEE Transactions on Image Processing*, vol. 11, no. 3, pp. 188–200, 2002.

[20] "CUDA toolkit documentation: CUFFT." Internet Source: docs.nvidia.com/cuda/cufft, 2016. Last Date Accessed: April 2016.

[21] "CUDA pro tip: Write flexible kernels with grid-stride loops." Internet Source: https://devblogs.nvidia.com/parallelforall, 2013. Last Date Accessed: April 2016.

[22] M. Harris *et al.*, "Optimizing parallel reduction in CUDA," *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.

[23] J. P. Lewis, "Fast template matching," in *Vision interface*, vol. 95, pp. 15–19, 1995.

[24] D. Padfield, "Masked FFT registration," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference*, pp. 2918–2925, June 2010.

[25] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar, "Optimizing data warehousing applications for GPUs using kernel fusion/fission," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pp. 2433–2442, IEEE, 2012.

[26] M. Harris, "Parallel prefix sum (scan) with CUDA." Internet Source: http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf, 2007. Last Date Accessed: May 2016.

[27] M. Harris, "An efficient matrix transpose in CUDA." Internet Source: http://devblogs.nvidia.com/parallelforall, 2013. Last Date Accessed: May 2016.

[28] M. Harris, "Using shared memory in CUDA C/C++." Internet Source: http://devblogs.nvidia.com/parallelforall, 2013. Last Date Accessed: June 2016.

[29] R. S. Jonnal, O. P. Kocaoglu, Q. Wang, S. Lee, and D. T. Miller, "Phase-sensitive imaging of the outer retina using optical coherence tomography and adaptive optics," *Biomedical optics express*, vol. 3, no. 1, pp. 104–124, 2012.

[30] R. D. Ferguson, Z. Zhong, D. X. Hammer, M. Mujat, A. H. Patel, C. Deng, W. Zou, and S. A. Burns, "Adaptive optics scanning laser ophthalmoscope with integrated wide-field retinal imaging and tracking," *JOSA A*, vol. 27, no. 11, pp. A265–A277, 2010.