Graduate School ETD Form 9
(Revised 01/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Brandon Andrew Shafer

Entitled
Real-time Adaptive-optics Optical Coherence Tomography (AOOCT) Image Reconstruction on a GPU

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

Is approved by the final examining committee:

John Jaehwan Lee

Russell C. Eberhart

Paul Salama

Lauren Christopher

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 on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): John Jaehwan Lee

Approved by: Brian King 04/25/2014

Head of the Department Graduate Program Date

REAL-TIME ADAPTIVE-OPTICS OPTICAL COHERENCE TOMOGRAPHY(AOOCT) IMAGE RECONSTRUCTION ON A GPU

A Thesis

Submitted to the Faculty

of

Purdue University

by

Brandon Andrew Shafer

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2014

Purdue University

Indianapolis, Indiana

For my Family who graciously put up with me.

ACKNOWLEDGMENTS

I would like to thank my friend and advisor Dr. Russell C. Eberhart for his encouragement and belief in my potential. I would also like to thank my major professor, Dr. John Lee, for his patience and guidance. I would especially like to thank Dr. Donald Miller and his team in Bloomington for their support and cooperation. The other members of my committee, Dr. Lauren Christopher and Dr. Paul Salama, were very helpful, and I would like to wish them thanks for their efforts.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

LIST OF ALGORITHMS

ABBREVIATIONS

CUDA	A parallel computing platform and programming model from NVIDIA.
SIMT	Single Instruction Multiple Threads.
GPU	Graphics Processing Unit.
GPGPU	General Purpose Computation on Graphics Processing Unit.
SM	Streaming Multiprocessor.
SMX	Streaming Multiprocessor for NVIDIA's Kepler Architecture.
ARMD	Age-related Macular Degeneration.
PCIe	Peripheral Component Interconnect Express, a standard of high-speed serial computer expansion bus.

NOMENCLATURE

Kernel	In CUDA, a parallel function running on GPUs.
Thread	An instance of a kernel.
Block	A grouping of threads implemented on an SM.
Grid	The set of blocks instantiated in a kernel launch.
Warp	32 threads that are executed in tandem.

ABSTRACT

Shafer, Brandon A. M.S.E.C.E., Purdue University, May 2014. Real-time Adaptive-optics Optical Coherence Tomography(AOOCT) Image Reconstruction on a GPU. Major Professor: John Jaehwan Lee.

Adaptive-optics optical coherence tomography (AOOCT) is a technology that has been rapidly advancing in recent years and offers amazing capabilities in scanning the human eye in vivo. In order to bring the ultra-high resolution capabilities to clinical use, however, newer technology needs to be used in the image reconstruction process. General purpose computation on graphics processing units is one such way that this computationally intensive reconstruction can be performed in a desktop computer in real-time. This work shows the process of AOOCT image reconstruction, the basics of how to use NVIDIA's CUDA to write parallel code, and a new AOOCT image reconstruction technology implemented using NVIDIA's CUDA. The results of this work demonstrate that image reconstruction can be done in real-time with high accuracy using a GPU.

1. INTRODUCTION

Adaptive-optics optical coherence tomography (AOOCT) is a technology that has been rapidly developing in recent years, increasing the capabilities of optical imaging to amazing heights. Using a complex array of adaptable mirrors, lasers, cameras, lenses, etc., researchers can scan the human eye and achieve detail down to the very rods and cones that give one sight. From *Adaptive Optics for Vision Science*:

> "The use of adaptive optics to increase the resolution of retinal imaging promises to greatly extend the information that can be obtained from the living retina. Adaptive optics now allows the routine examination of single cells in the eye, such as photoreceptors and leukocytes, providing a microscopic view of the retina that could previously only be obtained in excised tissue. The ability to see these structures in vivo provides the opportunity to noninvasively monitor normal retinal function, the progression of retinal disease, and the efficacy of therapies for disease at a microscopic spatial scale," [1, p. 11].

Adaptive-optics has allowed for the discovery of new properties of cone photoreceptors [2,3]. As with a lot of medical research, the end goal of technological advances of this nature is to aid in clinical diagnosis and treatment of patients. High resolution OCT can help with diagnosis and on-going treatment of glaucoma [4], neovascular age-related macular degeneration [5], macular hole [6,7], macular edema [6,8], among others. For example, in detecting glaucoma, the measuring the average thickness of certain quadrants of the retinal nerve fiber layer with OCT can be used to detect eyes with early glaucomatous visual field defects and eyes with early glaucomatous optic neuropathy [9]. This type of diagnosis can enable earlier detection and treatment. However, in clinical settings there exists constraints and considerations that may not exist in a research and/or lab environment. Namely, ophthalmologists and optometrists require an imaging technology that produces immediately available results to aid in diagnosis of patients. Tools that are provided to clinicians need to enhance their capabilities without hindering the throughput of their work.

AOOCT, by its nature, requires significant calculations to transfer the scanned data from the eye into human visible images for diagnosis. Although central processing units (CPUs) are increasingly powerful, the time required for these calculations using conventional sequential programming is prohibitive for a clinical environment. Whereas, graphics processing units (GPUs), designed in a very different way, lend to much better handling of certain kinds of problems that comprise a high degree of parallelism, e.g., working with matrices, images and video, data sets, etc.

The way that graphics processors work is different from a conventional CPU such as the Intel or AMD in PCs. CPUs are now very powerful devices that can handle numerous sequential instructions (step-by-step commands) and work at high clock rates. In recent years, those types of processors have moved to handling multiple instructions at once with the use of a few very powerful multiple cores where each core can handle its own stream of instructions. Instead of a few powerful cores, GPUs contain up to thousands of less powerful cores. Sets of cores are controlled by streaming multiprocessors (SMs). The cores under an SM operate in tandem, following the same instructions together each on their own set of data, what NVIDIA calls SIMT, i.e., single instruction multiple threads.

By using parallel programming in the form of NVIDIA's CUDA, we have been able to perform the necessary computations for image reconstruction in AOOCT, in real time. While some research has been done recently in using GPUs for OCT processing [10, 11], it is still a fairly new research area and as far as we know, this is the first time it is being applied to ultrahigh resolution AOOCT for use in the human retina.

This thesis is divided into the following chapters: Chapter 2 “Adaptive-Optics Optical Coherence Tomography,” Chapter 3 “CUDA,” Chapter 4 “AOOCT Processed With CUDA,” and Chapter 5 “Results.” In Chapter 2, we explain the technology of AOOCT and the process used to produce the final reconstructed images. Chapter 3 explains the basics of CUDA programming using CUDA-C. Chapter 4 explains how we use parallel programming to produce the process necessary to calculate volume images from AOOCT data in real time. Finally, Chapter 5 discusses the results of our work and suggestions for future progress.

2. ADAPTIVE-OPTICS OPTICAL COHERENCE TOMOGRAPHY

In the last couple of decades, enormous progress has been made in the area of retinal imaging *in vivo* (in the living human eye). One of these advances is in the area of adaptive-optics optical coherence tomography or AOOCT. Using ultra-high resolution spectral domain coherence tomography with adaptive-optics, researchers can obtain 3D resolutions as fine as 3 x 3 x 3 μm^3 [12, 13]. With this extraordinary resolution, individual rods and cones and even cells can be viewed in the living eye.

The evolution of this technology started with optical coherence tomography (OCT), first published with use *in vivo* in 1993 [14–16]. From the report "Optical Coherence Tomography" by Huang et al. comes the following description:

> "Both low-coherence light and ultrashort laser pulses can be used to measure internal structure in biological systems. An optical signal that is transmitted through or reflected from a biological tissue will contain time-of-flight information, which in turn yields spatial information about tissue microstructure" [14].

The first scanning was able to produce high spatial resolution of less than $2\mu m$, but the lateral resolution was limited by the beam diameter of the light to $9\mu m$ (of course this was in sample tissue not *in vivo*). Since 1997, adaptive-optics have been used to increase the resolution in OCT technology by correcting the ocular aberrations in real time using deformable mirrors. It was first developed to correct for atmospheric blur in ground-based telescopic systems, but is now a valuable tool in vision research [13].

To summarize, optical coherence tomography exploits the fact that different tissue reflects light differently in order to create a high resolution image. Adaptive-optics then corrects for diffraction caused by the eye's lens and cornea. Even though a type of CCD camera is used to obtain the “image” from the system, the image must be processed in order to reconstruct the 3 dimensions of the retina and, subsequently, become useful for a clinician. The following subsections describe the steps taken to acquire the image and calculate the final result in our research. In describing AOOCT images in this paper, the terms A-line, B-scan, and volumes are used as shown in Fig. 2.1. We process the image data a B-scan at a time.

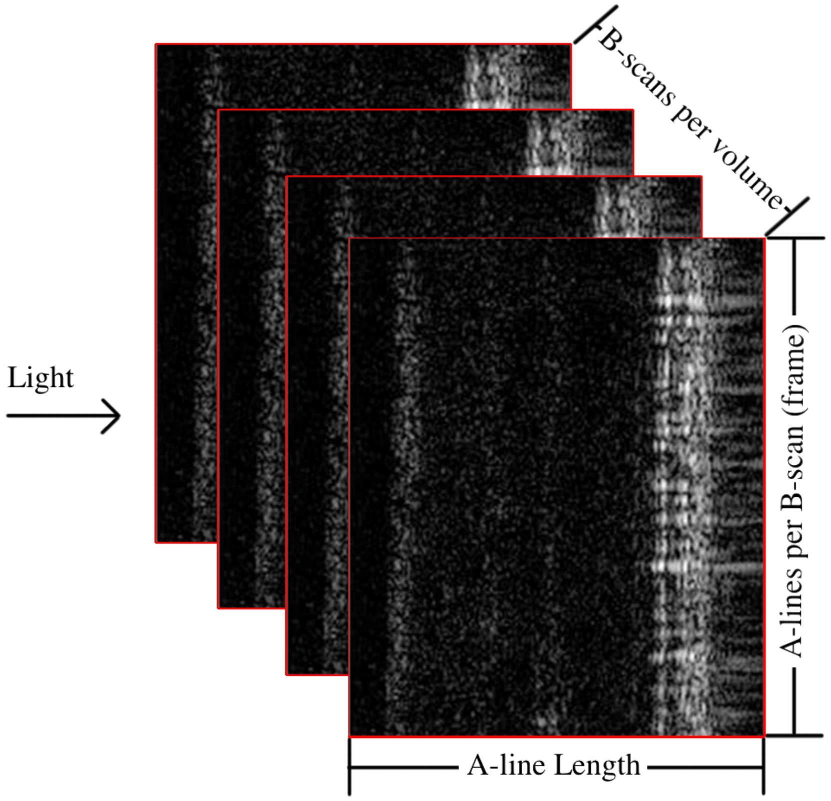


Fig. 2.1.: Volume AOOCT images.

2.1 System

An AOOCT system consists of laser(s), deformable mirror(s), camera(s), and assorted optics and optical devices. The complete setup is beyond the scope of this thesis, but a good resource can be found in *Adaptive Optics for Vision Science* [1].

Optical coherence tomography allows for scanning of very fine resolution, but with the human eye, the imaging is distorted by the cornea and the lens. Prior to this research, in our ignorance of the eye, we had only thought of the lens as being the sole optic in the eye. However, the cornea also affects the light that enters the eye. Interestingly, in a normal eye, the cornea and the lens on their own have greater optical aberration than they have together. When correcting vision, such as in LASIK, the cornea is corrected to match the lens, not to make the cornea optically perfect on its own, which would make the vision ironically worse.

Adaptive-optics are used to correct for the optical aberrations introduced by the cornea and the lens. Either by an open or closed feedback loop, using a reference signal and an interferometer, deformable mirrors are adjusted to compensate for the aberrations. In Dr. Miller's lab, with which we have worked very closely, two deformable mirrors are used, one with low resolution but large range for coarse adjustment and one with higher resolution and lower dynamic range for fine adjustment.

After initial setup, a laser is shone into the eye onto the retina. The returning light is captured using a CCD camera. Both the scanning laser and the camera are controlled via a computer, maintaining synchronization. Once in the computer, the data needs to be processed to become understandable images. The processing, described in Section 2.2, can be performed at runtime or after the fact. However, runtime is preferable, and was our goal in this research, to enable the system to be fast enough for clinical use, and to provide real-time visual feedback to the human operator of the system to make adjustments for better results. One of the limitations of imaging the eye is the difficulty in the subject keeping still and the minor discomfort of a subject in placing the head in position to be scanned. Faster feedback in the form of real-time imaging helps the operator to calibrate the system faster and to see when a recording needs to be redone from movement.

2.2 Processing Steps

AOOCT images are not readily apparent upon capture. Like a magnetic resonance image in a hospital, the image must be processed to become useful to human eyes. The steps of processing an AOOCT image after scanned spectral data is input into the system is as follows, and the details of each step are explained in the following subsections.

1. DC component subtraction.
2. Pad the A-lines to predetermined width.
3. Map the image to k-space.
4. Compensate for dispersion.
5. Fast Fourier Transform(FFT) to convert the image to time domain.
6. Crop and convert to pixels.
7. Final visualization.

2.2.1 DC component subtraction

When the A-lines are initially scanned, there exists a strong peak near zero frequency in the spectra. There are a couple reasons we want to remove that peak. First, during calibration, which is prior to but similar to our reconstruction process, the highest peak of a physical structure is determined. It is necessary to remove the DC peak because it interferes with this detection. Since our data also uses information from the calibration, it is necessary to match that process. Second, if the DC peak was not removed, in the final image it would present as a constant brightness and interfere with the user being able to descern actual physical structural information from the DC signal.

To subtract the DC component out of the OCT signal, an averaged A-line spectrum is calculated from all of the A-lines of a B-scan as seen in Eq. 2.1.

$$\bar{A}_i = \frac{1}{M} \cdot \sum_{j=0}^{M-1} A_{ij} \tag{2.1}$$

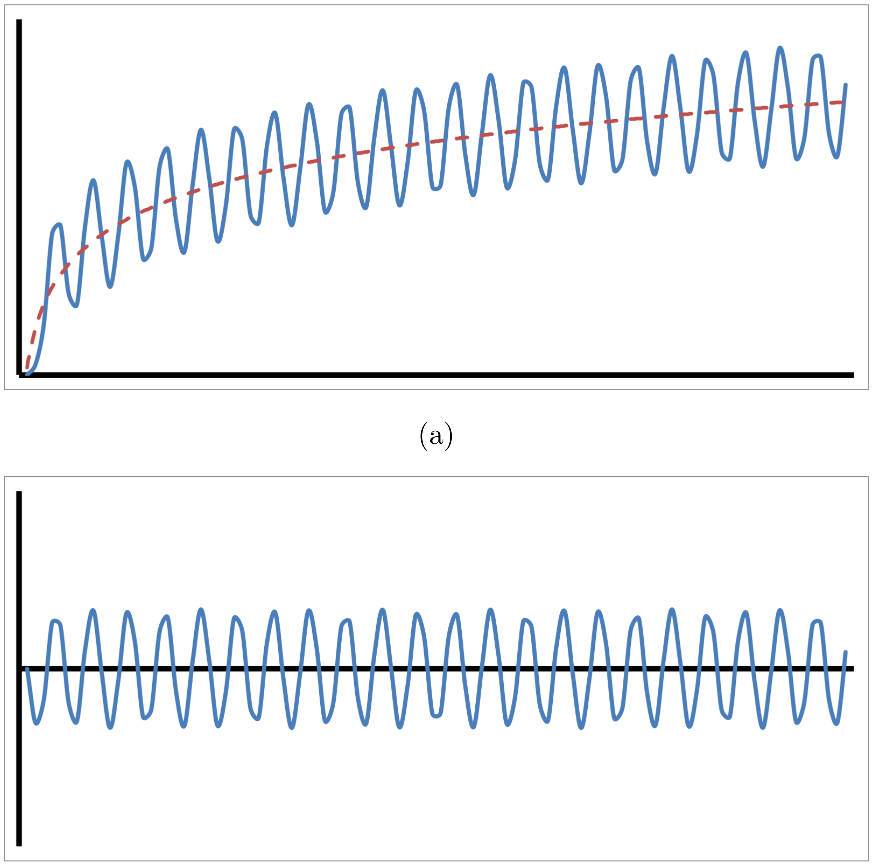
where i is an element of A-line, j is the row in the B-scan, and M is the number of A-lines in the B-scan. Then the averaged component is subtracted from each element in the B-scan as seen in Eq. 2.2.

$$A\text{new}_{ij} = A_{ij} - \bar{A}_i, 0 \leq j < M, 0 \leq i < N \tag{2.2}$$

The overall effect is visualized in Figure 2.2.

2.2.2 Zero-padding

Zero-padding the incoming A-lines serves a couple purposes. It allows us to work with sizes that are a power of two. This allows for faster subsequent FFTs and is more conducive to GPU operations. Also, zero-padding, using Fourier transforms as we do, acts as a form of interpolation. We cannot add data that is not present to begin with, but this allows us to use more pixels with smaller frequency bin sizes, creating a smoother image. For example, an input signal with A-lines that are 832 pixels wide is zero-padded up to a size of 4096 pixels wide. The same information exists in the 4096 wide version as in the 832 wide version, but displaying the resultant image in 4096 pixels will allow for a smoother, easier to understand image. Padding the signal is a four step process. First, the initial signal is padded on both ends of the A-line to have a size of a power of two to better facilitate the GPU as cuFFT runs faster when the size is a power of two. The next three steps involve passing the signal through an FFT, adding padding, and bringing it back through an IFFT. This spreads the signal over the whole width we are allotting. It is akin to using a spread-spectrum signal in communications. The first FFT is performed as shown in Equation 2.3. Because the signal is real-valued, the transform $\mathcal{F}(x)$ is hermitian. This can be and is exploited to increase processing speed by decreasing storage size.



(a)

(b)

Fig. 2.2.: DC Subtraction from each scanned signal: The solid oscillating lines represent a single A-line. Image (a) is an individual A-line superimposed on the dotted line representing an averaged A-line, $\bar{A}_i$, from the entire B-scan. (b) The averaged data is subtracted out of the individual A-line.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} \tag{2.3}$$

Third, each A-line is padded with zeros again as follows. If the full A-line were seen, the padding would be in the middle, but since we are using hermitian symmetry (discussed later in Section 4.4), only one side of the A-line is stored and padding only needs to be applied to the end. Fourth, perform an inverse FFT as follows.

$$X_k = \frac{1}{N} \cdot \sum_{n=0}^{N-1} x_n \cdot e^{i2\pi kn/N} \tag{2.4}$$

The signal going into the first FFT and the signal after the inverse FFT are real-valued. Currently, if the initial A-line width is less than 2048, it is padded to 2048 in step one, and finishes at 4096 after step four. If the initial A-line width is greater than 2048, the numbers become 4096 and 8192, respectively. These numbers were chosen by Dr. Miller's team in Bloomington for the resulting images to have appropriate resolution.

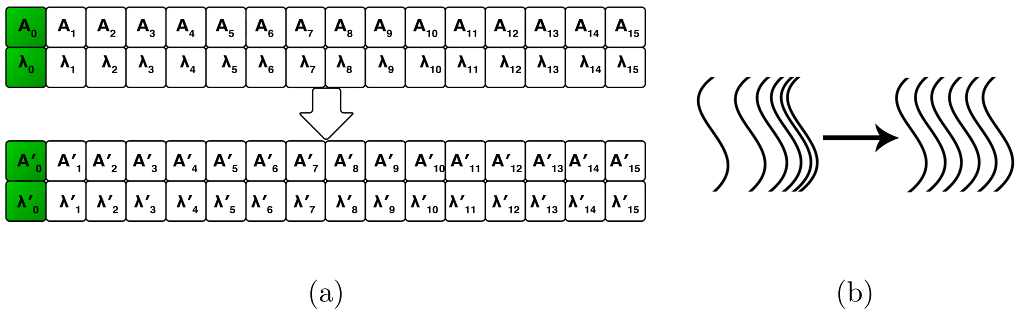
2.2.3 K-space interpolation

Each pixel in an A-line corresponds to a wavelength of light. When captured, the wavelengths associated with each A-line are not evenly spaced. In order to create an evenly spaced final image, first the A-line spectra are realigned from λ-space to k-space, where $k = 2\pi/\lambda$, and then the samples need to be equally spaced in k-space. In order to do this, a calibration file is needed with the current wavelengths that are associated with the A-line spectra. The first and last wavelengths are converted to k-space as shown in Eq. 2.5 and Eq. 2.6.

$$k_{min} = (2 \cdot \pi)/\text{wavelength}[0] \tag{2.5}$$

$$k_{max} = (2 \cdot \pi)/\text{wavelength}[\text{end}] \tag{2.6}$$

A vector is then created with evenly spaced values with k_{min} and k_{max} as the beginning and ending elements. All of the values of the new vector are converted back to λ-space so that the current A-line spectra can be interpolated to the new evenly spaced, in k-space, values as shown in Figure 2.3. We then use a simple linear interpolation to complete the process. Linear interpolation is used in this work in the interest of faster computation time, but in future works better forms of interpolation (e.g., spline interpolation) can be explored.



A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15
λ0 λ1 λ2 λ3 λ4 λ5 λ6 λ7 λ8 λ9 λ10 λ11 λ12 λ13 λ14 λ15
A'0 A'1 A'2 A'3 A'4 A'5 A'6 A'7 A'8 A'9 A'10 A'11 A'12 A'13 A'14 A'15
λ'0 λ'1 λ'2 λ'3 λ'4 λ'5 λ'6 λ'7 λ'8 λ'9 λ'10 λ'11 λ'12 λ'13 λ'14 λ'15
(a)
(b)

Fig. 2.3.: (a) shows an A-line with pixels of $A_0, A_1...A_{15}$ and wavelengths $\lambda_0, \lambda_1...\lambda_{15}$ transformed to new values $A'_0, A'_1...A'_{15}$ associated with new wavelengths that are evenly-spaced in k-space, $\lambda'_0, \lambda'_1...\lambda'_{15}$. (b) illustrates the uneven wavelengths transforming to evenly-spaced wavelengths.

2.2.4 Dispersion compensation

OCT images are often blurred from an optical effect called dispersion [17, 18]. The light that enters the eye is dispersed because of the refractive index of the tissue. The adaptive-optics help to counteract that effect, but do not completely eliminate it. One reason for this is that the dispersion affects different wavelengths differently. According to Cense et al., the relation between multiple orders of dispersion and the phase $\theta(k)$ can best be described by a Taylor series expansion:

$$\begin{aligned}\theta(k) = \theta(k_0) + \left.\frac{\partial\theta(k)}{\partial k}\right|_{k_0}(k_0 - k) + \frac{1}{2}\cdot\left.\frac{\partial^2\theta(k)}{\partial k^2}\right|_{k_0}(k_0 - k)^2 \\ + \ldots + \frac{1}{n!}\cdot\left.\frac{\partial^n\theta(k)}{\partial k^n}\right|_{k_0}(k_0 - k)^n\end{aligned} \tag{2.7}$$

with λ_0 the center wavelength and k_0 equal to $2\pi/\lambda_0$ [18].

The third term represents second order dispersion. It is manifested along the A-lines, and to remove the blur, each element in an A-line is multiplied by a complex phase term determined during calibration of the system. The complex phase terms are found in the human eye using a well-reflecting reference point, the center of the fovea. After dispersion compensation, the resulting B-scans are complex-valued.

2.2.5 Fast Fourier Transform (FFT)

All of the prior steps have been preparing the data for this reconstructive step. This FFT will reconstruct the data into an actual retinal volume image. The FFT in this step is a 1D FFT along each A-line. The B-scan is complex-valued from the previous step, and resulting image frame will be complex-valued as well. The FFT is the same as in Equation 2.3

2.2.6 Final conversion to image

At this point, the B-scan is reconstructed into a complex-valued image. There are several more steps to finish formatting the image for display, and there are some optional steps that help optimize the image for the end user.

1. Crop the image.
2. Convert complex numbers to intensity.
3. Convert to log scale (Optional).
4. Normalize into pixels.

The cropping of the image disposes of unnecessary parts of the A-lines. The beginning of the A-lines often contains large optical artifacts due to being near the coherence gate of the OCT system. The end of the A-lines often contains little useful information. Prior to finding the intensity values, the image is cropped to exclude those regions of the A-lines. Then, the complex numbers are converted to an intensity value according to Eq. 2.8.

$$I_{ij} = \Re(B_{ij})^2 + \Im(B_{ij})^2 \tag{2.8}$$

where i is the column, j is the A-line position in the B-scan, B is the B-scan, and I is the resulting image. If a log-scaled image is desired, the log is then calculated as in Equation 2.9.

$$I_{\text{log}ij} = 10 \cdot \log_{10}(I_{ij}) \tag{2.9}$$

where I_{log} is the resulting log-scaled image, i is the column, j is the row in the B-scan, and I is the image resulting from Equation 2.8. The images are then normalized and fit into unsigned character bytes of values of 0-255. The resulting image is 8-bit grayscale.

$$N_{ij} = (\frac{I_{ij} - I_{\text{min}}}{I_{\text{max}} - I_{\text{min}}}) \cdot 255 \tag{2.10}$$

where I is either the resulting image from Equation 2.8 or 2.9, whichever is desired, I_{min} and I_{max} are the minimum and maximum of the individual pixels in the B-scan, respectively, i is the column, j is the row in the B-scan, and N is the resulting normalized image. The minimum and maximum can be calculated automatically or can be set by a user to make the resulting figure easier to visually understand. When the minimum is set manually, data that is under the minimum will be clamped to zero, and likewise for a manually set maximum, data that is greater than the maximum will be clamped to 255.

Figure 2.4a shows what a linear B-scan looks like once it is complete. Figure 2.4b shows that same figure as a log scale.

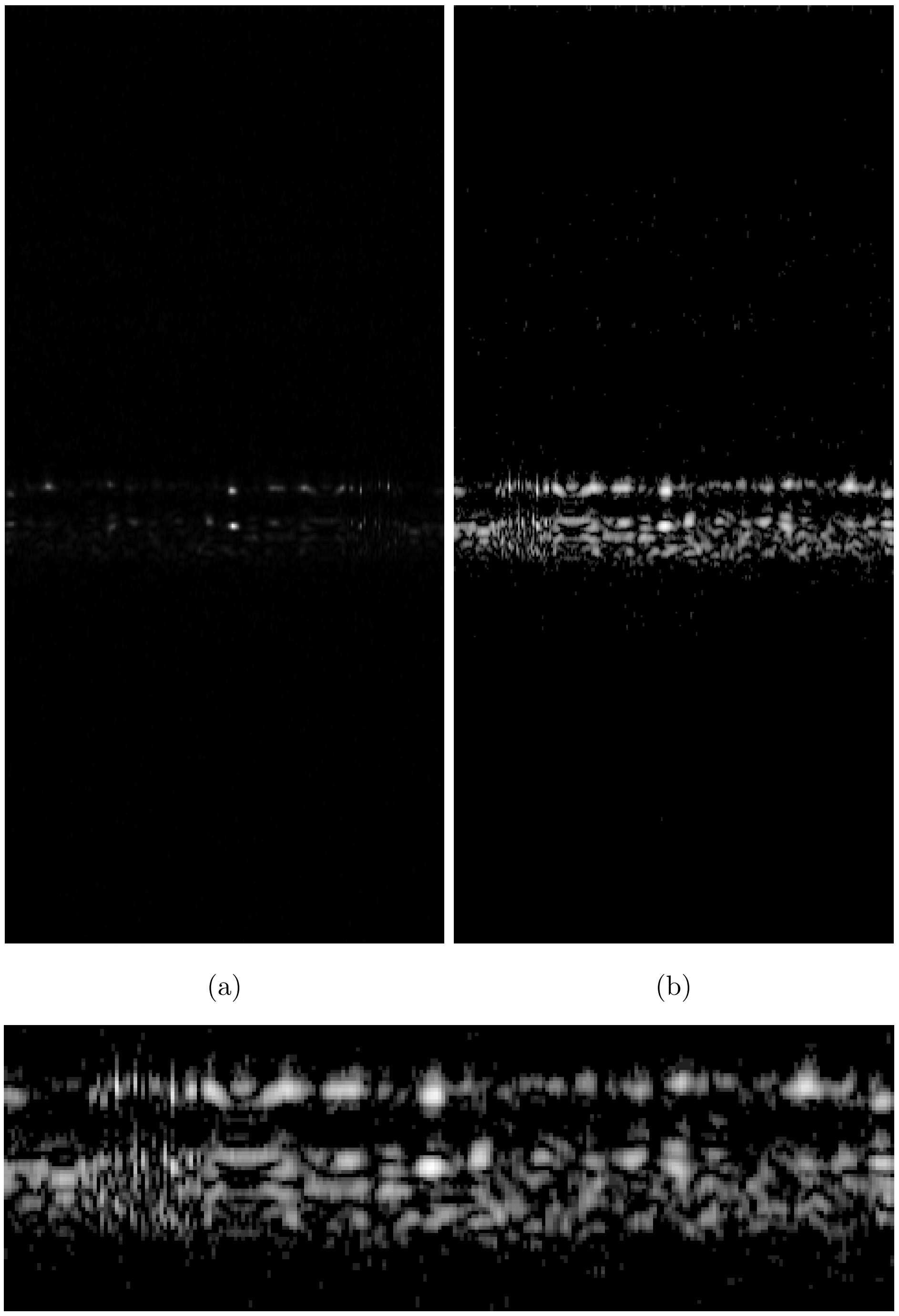
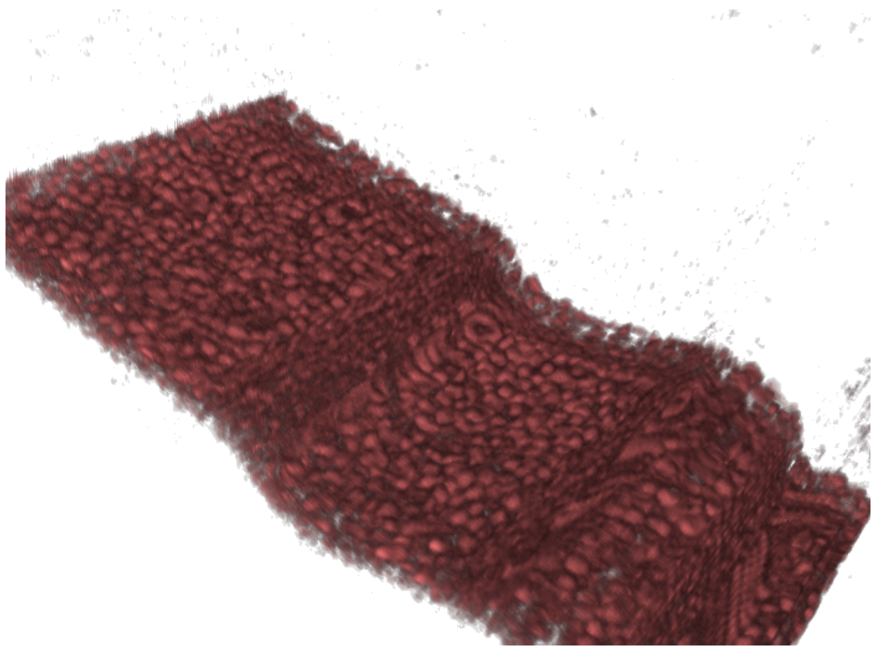
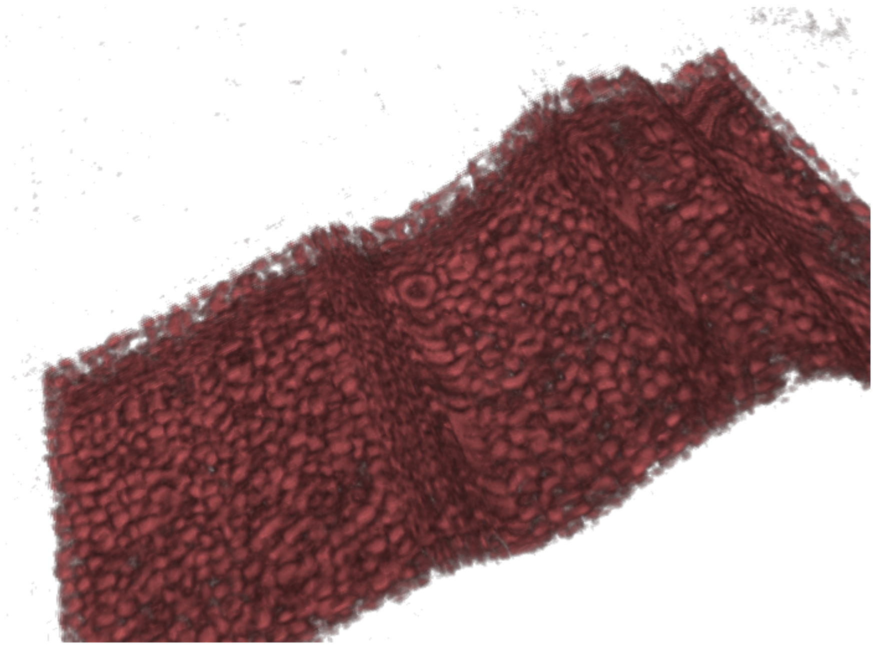


Fig. 2.4.: A processed B-scan where (a) is a linear version, (b) is a logarithmic version, and (c) is cropped from (b)). These images were produced using our software from data provided by Dr. Miller's lab.



(a)



(b)

Fig. 2.5.: 3D projected volume of processed B-scans from two different angles. These images were produced using the software ImageJ using reconstructed images from our software.

3. CUDA

Almost all processors use the Von Neumann approach to computing where a processor fetches an instruction from memory, decodes that instruction, and performs the operation of that instruction [19, p. 12]. Conventional CPU processors have become very powerful over the years, increasing clock speed and throughput. Nowadays the move has been also to more cores. Two, four, eight, etc. cores in a processor allow for many more instructions to be processed and increases throughput. However, the programmer must make use of the multiple cores, a non-trivial proposition, to use this expansion of capability to enhance his or her own programs.

The GPU has come in favor over the last few years for general scientific computing for a several reasons. To start with, GPUs come at problems in a way more akin to the way past super computers addressed them than to a modern CPU; that is, they utilize many cores to break a problem into many smaller parts to be worked. The overall clock speeds of GPUs are much lower than the powerful CPU of most modern PCs. For example, the NVIDIA Titan (the most advanced NVIDIA gaming GPU) runs at a clock speed less than 1GHz [20], while most CPUs run around 3 GHz. However, the GPU uses multiple supervisory cores, and thousands of CUDA cores, to make quick work of problems that could take much longer using a CPU. Secondly, GPUs are far less costly than supercomputers, and even Multi-Nodal computer systems. At the time of this writing a single multicore processor can cost over a thousand dollars. A Titan, costs approximately a thousand dollars, and a Tesla K20, the card used in this research, around three thousand. A GPU can be put in a standard desktop PC and even some laptops, and if more power is needed more cards can be added via the PCIe bus. Thirdly, with the advent of CUDA C, from NVIDIA, writing programs for GPUs has become far more accessible to scientists. In this chapter we discuss the structure of CUDA GPU cards and the basics of how to program with them.

3.1 Parallelism

CUDA programming starts with a normal program like any other in C++ or C (CUDA-C is the fundamental path to coding for CUDA, but there are extensions available for Python and other languages). However, actual computer code will be divided in between work done on the GPU and work on the CPU. Which code is for the GPU and which is for the CPU is explicitly written into the program. Code that is for the GPU comes in the form of kernels. A kernel is akin to a function or method in C/C++. There are arguments and the kernel, like a function, can contain its own variables. A kernel is invoked from other code. However, there are some big difference between a standard function written for a CPU and the kernel written for the GPU. A kernel can be called in a way that it is run on many cores at once. While more is said about the GPU hardware in Section 3.2, we also discuss it here.

While CUDA was an evolution from prior GPUs that were primarily for graphics, but which had programmable properties, the first full-fledged architecture from NVIDIA which allowed for General Purpose computing was the Fermi architecture. In Fermi, the cards are designed with a number of Streaming Multiprocessors (SMs). Each SM is made up of 32 cores with local registers, 64KB of SRAM between cache and local memory, four special function units (SFUs), 16 load/store units for memory access, and can perform integer and floating point operations [21]. As shown in Figure 3.1, the cores are divided into two sets of 16. Along with the SFUs and the load/store units there are four execution blocks in an SM. In any one clock cycle, 32 instructions from either one or two warps (a warp is a bundle of threads, with each thread being an instruction), can be issued to the execution blocks. As shown in Figure 3.2, the Kepler SMX, the successor to Fermi, has expanded capabilities and is much more powerful. An SMX has more cores, double precision, more load/store units, more SFUs, and greater memory than the Fermi SM.

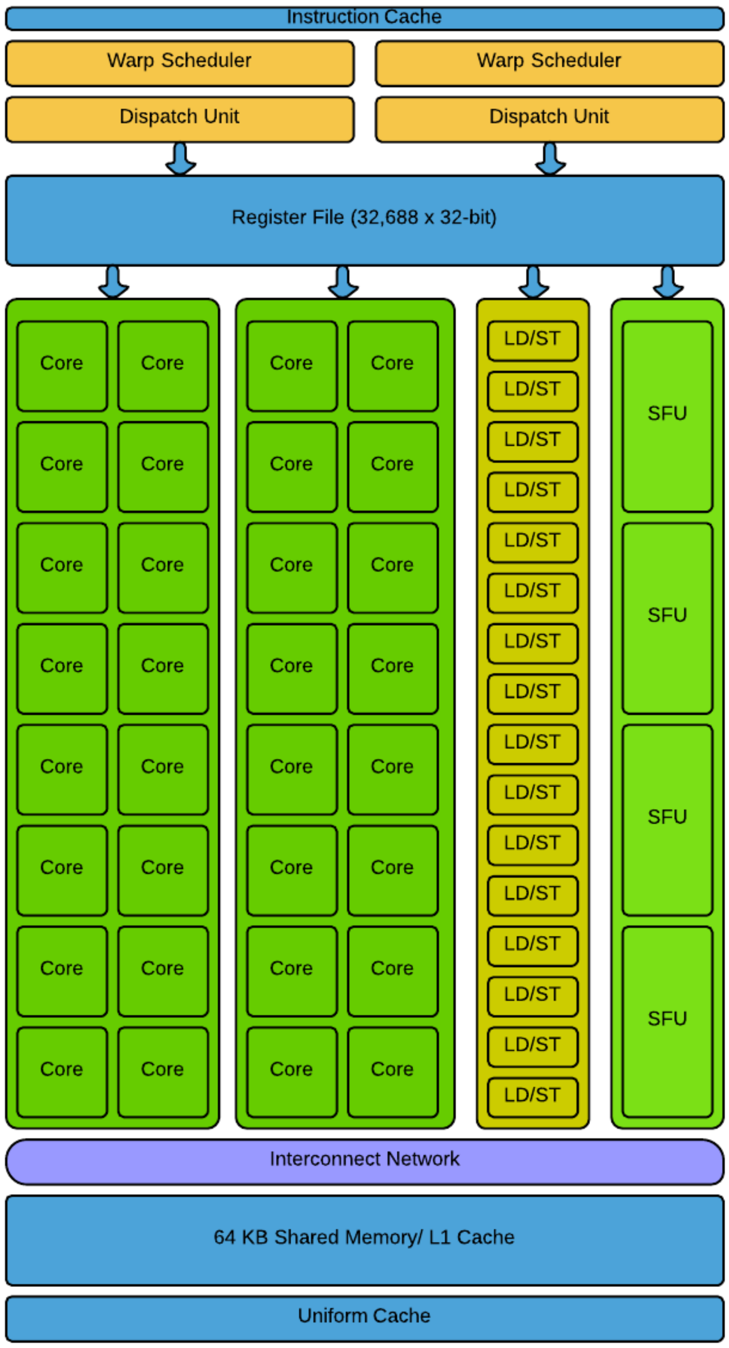


Fig. 3.1.: Fermi streaming processor [21]

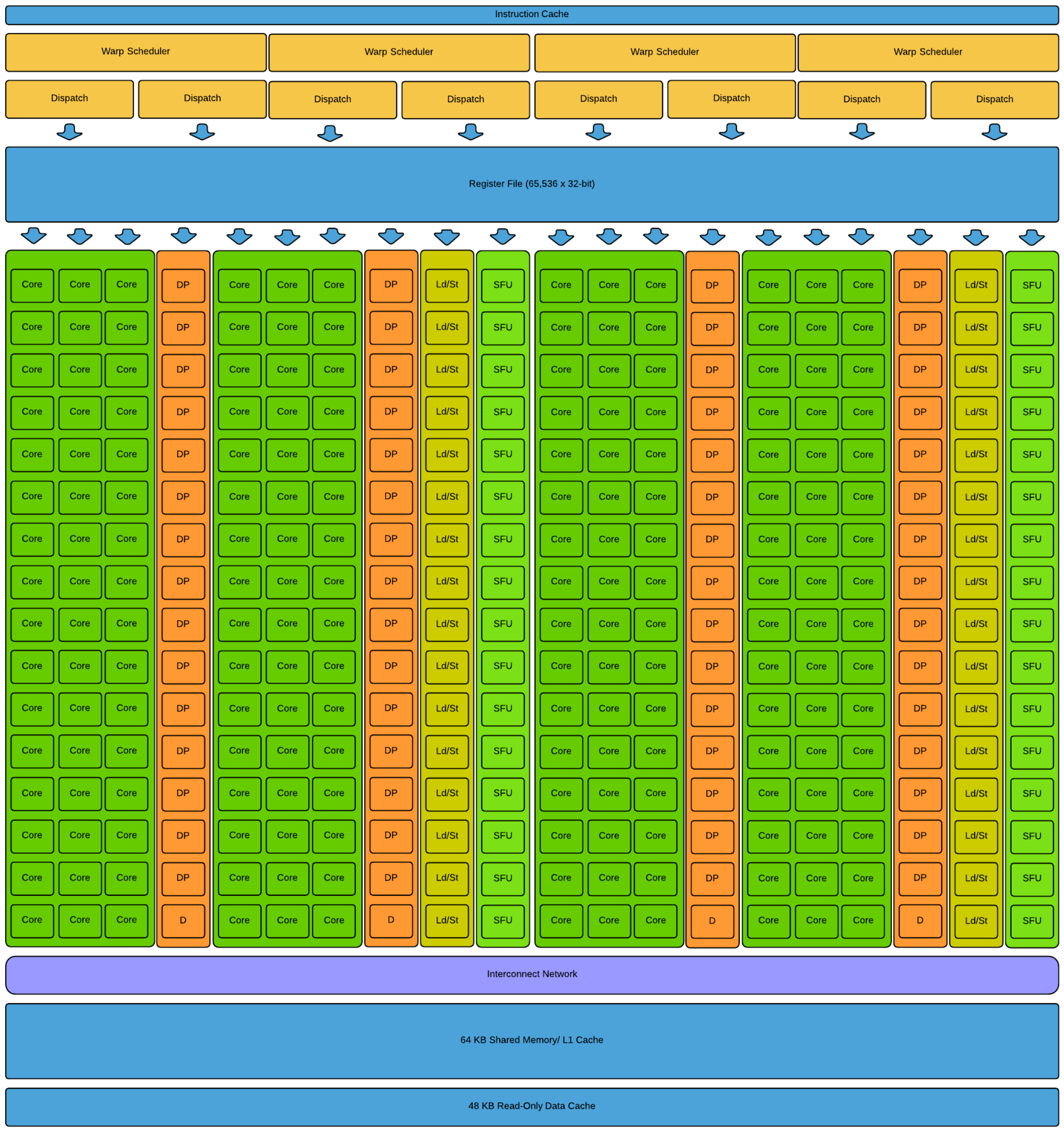

Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP
D
Ld/St
SFU
Interconnect Network
64 KB Shared Memory/ L1 Cache
48 KB Read-Only Data Cache

Fig. 3.2.: Kepler SMX streaming processor [22]

CUDA operates under the SIMT model which stands for Single Instruction Multiple Thread. When an instruction is issued in a warp of threads, a full warp being 32 threads, this warp would encompass all the execution blocks of cores in the SM.

The same instruction would be executed on each core, whereas each core operates on its own thread. Each thread will be written to use its own data, so that the threads in a warp are operating in parallel.

Listing 3.1: A vector addition example.

```
1 __global__ void vecAdd(float* C,float* A,float* B)
2 {
3   int idx = threadIdx.x;
4   C[idx] = A[idx] + B[idx];
5 }
```

Using the simplest of examples, vector addition, in Listing 3.1 a simple kernel is written. This code is very simple with just two lines. On line number three, there is an internal variable idx assigned from threadIdx.x. When a kernel is launched, each thread has certain properties that can be exploited. Those properties give the programmer an idea as to which specific thread is being worked on. That is not to say that the programmer knows exactly where the thread will be executed in hardware, or the order in which the thread will be executed in hardware. Those are issues handled by a scheduler and can be different every time the program is run. This does give the programmer the ability to specify different data for each thread to work on. For illustration, say this kernel is launched as one warp of 32 threads. That warp will be put onto a single SM, where each core will handle one thread. Line number three uses the thread information (threadIdx is defined by CUDA and is not a user defined variable) to reference the position in the vector where it will handle information. So for example, there will be a core with thread number 16. On that core line number three will assign a 16 to idx. Then line number four will be executed as C[16] = A[16] + B[16]. Compare that to thread number 8, which will be executed as

C[8] = A[8] + B[8]. The entire warp progresses through the code at the same time, each core taking the same instruction on its own thread, but each instruction can reference its own registers or memory.

Threads are grouped as warps that run on the cores of the SM, but when calling a kernel, the threads are grouped in blocks. A block is made up of a number of warps, with possibly more threads than can be run at a single instant, but few enough that all of the memory associated with the block can fit on a single SM [23, p. 8]. On current CUDA cards, a block can have up to 1024 threads. The kernel can be broader than just a single block however. Multiple blocks of the same dimensions can be called to operate the same kernel. In Listing 3.2, the kernel call for Listing 3.1, the threads are in a single block. In Listing 3.3, the original kernel has now been adapted to work on multiple blocks.

Listing 3.2: A vector addition example kernel call.

```
dim3 blocks (1,0,0);
dim3 threads (32,0,0);
vecAdd<<<blocks,threads>>>(C,A,B);
```

Listing 3.3: A vector addition example using multiple blocks.

```
__global__ void vecAdd(float* C,float* A,float* B)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  C[idx] = A[idx] + B[idx];
}

dim3 blocks (5,0,0);
dim3 threads (128,0,0);
vecAdd<<<blocks,threads>>>(C,A,B);
```

CUDA allows for up to three dimensions in the references in blocks and threads. This makes referencing in matrices fairly straightforward. In Listing 3.4, the vector addition example is expanded to work on a matrix. The differences between this example and the previous example are the dimensions of the blocks, and the referencing in the kernel.

Listing 3.4: A vector addition example using multiple dimensions.

```
__global__ void matAdd(float* C,float* A,float* B)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int idy = blockIdx.y * blockDim.y + threadIdx.y;
  int gid = idy * gridDim.x * blockDim.x + idx;
  C[gid] = A[gid] + B[gid];
}

dim3 blocks (5,10,0);
dim3 threads (32,4,0);
vecAdd<<<blocks,threads>>>(C,A,B);
```

Figure 3.3 shows how a two dimensional matrix can be represented in memory. The matrix is stored row by row, so that Row 0 comes first then Row 1, etc. In Listing 3.4, the individual elements are referenced in this way. The y dimension is multiplied by the length of a row and then that result is added to the x dimension to obtain the dereferenced element. In the same way, when referencing an element across blocks, it is necessary to multiply the blockIdx (the id of a block) by the blockDim (the number of threads in a block) and then add the threadIdx to get to the proper element.

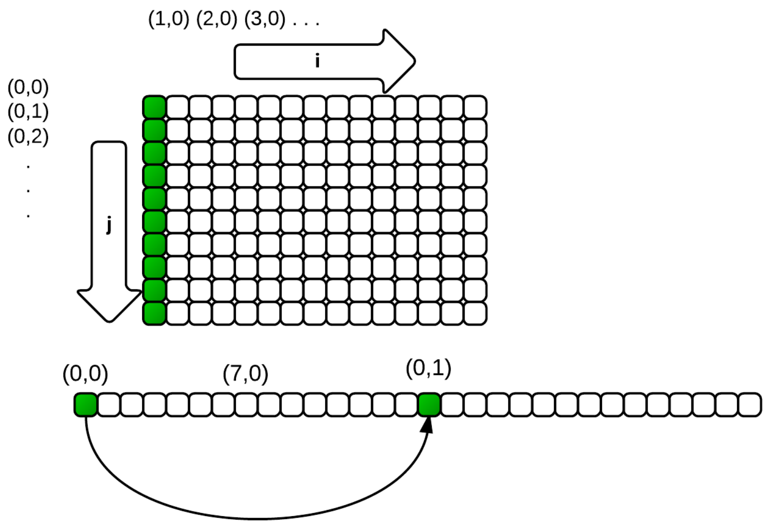


Fig. 3.3.: Matrix storage structure

When writing parallel code for CUDA, it is also necessary to examine the relationship between threads in a block and between threads in different blocks. Threads in the same block can share some local memory (NVIDIA calls it shared memory). This is important because shared memory is much faster than global memory. In the same way that cache is much faster than the memory in a CPU, the shared memory of an SM in a GPU is much faster than the general global memory of the GPU. While CUDA makes shared memory an option that can be used in a kernel, the threads themselves can be on different warps and be executed at different times. In instances where threads depend on data written by other threads, it is necessary to synchronize using __syncthreads(). This synchronization can come at a cost of computation time, so threads that do not require communication are much preferred if possible. Also, the order of thread execution is not something that can be known and relied upon ahead of time.

While threads can communicate with threads in the same block, cross-block communication is not reliable. There are operations called Atomic Operations provided by CUDA that enable a kernel to do a limited set of operations on shared and global data that prevent race conditions. For example, two instances of a kernel that reside in different blocks on different SMs could both write to the same location in global memory and both writes will be honored and not interfere with each other when using Atomics. However, the order of the write operations cannot be known ahead of time. Furthermore, atomics do come at a cost; the atomic operations take longer than normal read/write operations, and when two threads are in conflict, one will have to wait, meaning that whole warp will be delayed.

3.2 Hardware

The newest iteration of CUDA architecture is the Kepler architecture. The GPU used for our research is the Tesla K20. Kepler has numerous advances over the previous generation that enable faster run times and better overall throughput. For example, the new SMX streaming multiprocessor of the Kepler architecture is more powerful, programmable and energy efficient than the previous generation. While each SM has 32 cores , four special function units, and 16 load/store units for memory access, the new Kepler SMXs have 192 cores, 64 double precision units, 32 special function units, and 32 load/store units for memory access [22]. While not used in our research, dynamic parallelism also offers the ability to call a kernel from a kernel. In instances where a subsequent kernel needs information from a previous kernel before launching, that subsequent kernel can be called from a GPU with dynamic parallelism, whereas before it could not. Keeping that decision making and kernel launches on the GPU can offer greater throughput and speed in the circumstances that require it. NVIDIA is constantly innovating new capabilities with their GPUs, besides the impressive increase in computing power via better hardware.

Table 3.1: Fermi and Kepler GPU Capabilities [22]

	Fermi GF100	**Fermi GF104**	**Kepler GK104**	**Kepler GK110**
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	$2^{16}-1$	$2^{16}-1$	$2^{32}-1$	$2^{32}-1$
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

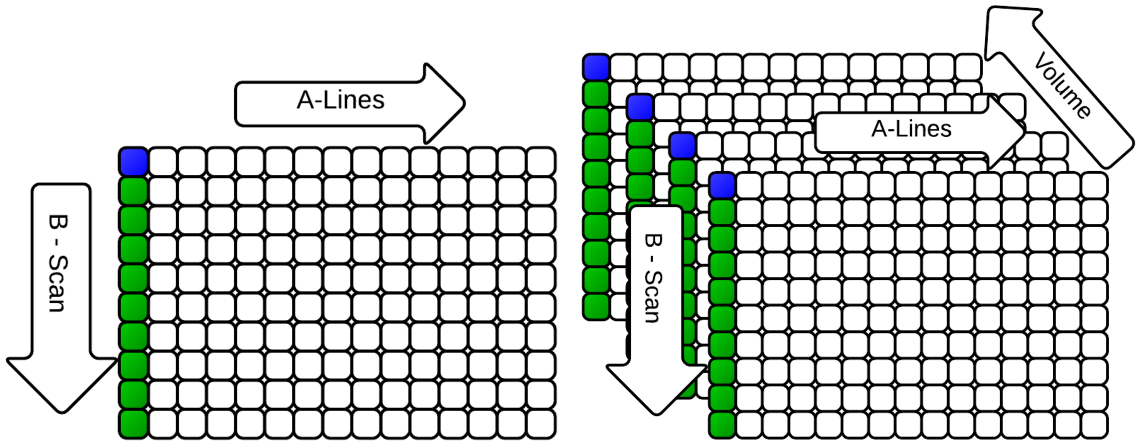
Besides the Tesla K20c GPU card, the machine that this research was performed on also had 64 gigabytes of DDR3 RAM, an Intel Xeon processor, a Quadro K2000D GPU, and a large 500 Gb SSD hard drive.

4. AOOCT PROCESSED WITH CUDA

Adaptive-optics optical coherence tomography (AOOCT) has enabled incredible resolutions in scanning the living eye that were not possible before. AOOCT has allowed the capture of volume images of structures in the retina, previously un-viewable except by using microscopes. Now, with AOOCT, a number of clinical conditions are being studied, including age-related macular degeneration (ARMD), hereditary retinal dystrophies, retinopathy of prematurity, and optic neuropathies [13]. Still, the major drawback of this technology is that the result is computed, and the computations take significant time. In order to push the technology to the next level, introduction into regular clinical use, the calculation time needs to be within seconds instead of hours. It is in this regard that CUDA parallel programming can be of important assistance to this field. Algorithm 4.1 shows how a B-scan progresses through the GPU.

Algorithm 4.1 B-scan Reconstruction.

```
1: for all $i$ such that $0 \leq i <$ number of B-scans do
2:     Convert incoming spectral data from 16-bit unsinged integer to float
3:     Subtract DC component from B-scan
4:     Pad A-lines
5:     Align to $k$-space
6:     Compensate for dispersion
7:     1D FFT of each A-line
8:     Calculate intensity
9:     Normalize
10:    Move to next available CUDA stream for next B-scan
11: end for
```



(a) (b)

Fig. 4.1.: Each row of the storage array is an A-line. A-lines then combine to make a B-scan. In memory, rows are stored contiguously, i.e., A-line 1 is stored then A-line 2, A-line 3, etc. as in (a). Green squares mark the beginning of an A-line, and blue squares mark the beginning of a B-scan. In this program, B-scans are handled one at a time, but once finished are stored as in (b).

As mentioned in Section 3.1 and shown in Figure 3.3, it is important to realize how the data is stored in memory. This is particularly true for usage on a GPU as hiding memory latency is an important factor in the overall speed of a CUDA kernel. In the following sections, data will be handled as a B-scan at a time, and like the matrix in Figure 3.3, each B-scan will be stored as in Figure 4.1a.

4.1 Converting Incoming Data

Line 2 from Algorithm 4.1 converts the incoming data from 16-bit unsigned integer to float. The spectra data as scanned by the camera is stored in memory as unsigned 16-bit integers. However, to perform the calculations needed in the reconstruction it is much better to use floating point numbers. The incoming data is copied to the GPU RAM as an array of unsigned 16-bit integers. In a very simple GPU kernel, each pixel is cast as a single precision float. A single precision float is stored in four bytes. Then the pixel, as a single precision float, is stored in global memory on the GPU.

4.2 DC Subtraction

Line 3 from Algorithm 4.1 is to remove the DC component of the incoming A-line signals for the entire B-scan. The way to accomplish this is to calculate the average for each point in all of the A-lines of a B-scan, as seen in Figure 4.2, and then subtract the average from all of those points.

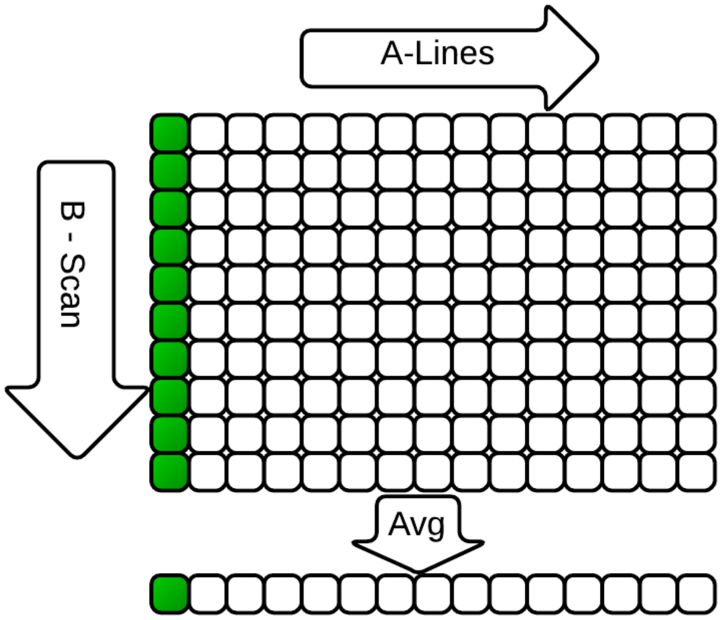


Fig. 4.2.: Averaging B-scan for DC subtraction.

When first approaching this problem, the most straightforward method was to write a kernel where a thread would loop through a column, accumulating a summation, divide by the number of elements, and loop through again subtracting the average from each element. We initially implemented this technique.

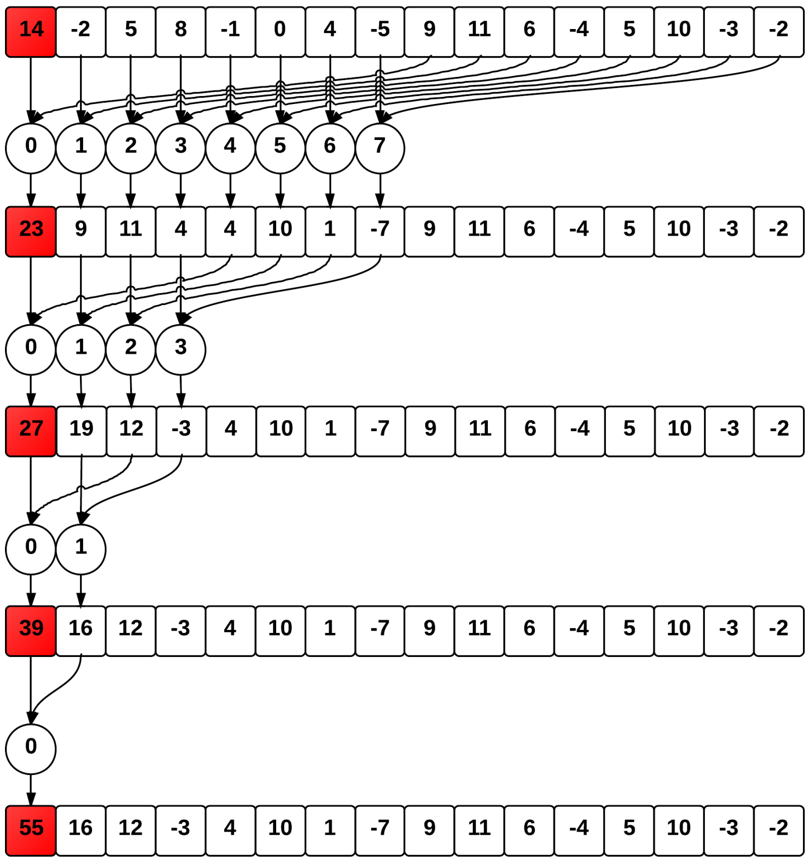


Fig. 4.3.: Parallel reduction.

However, summing the elements linearly like this seemed very inefficient compared to another summation technique called Reduction. Linearly adding and subtracting through the columns takes on the order of $2N$ steps per thread. In reduction, the elements are added together in steps similar to a "Divide and Conquer" recursive technique in traditional programming [24]. Figure 4.3 shows how the elements are combined. Generally, this allows for reducing the number of steps to $\log_2 N$ per warp.

However, to use reduction for DC subtraction, the B-scan needs to be transposed first. The reason for this is in the way that memory is accessed by the GPU. If the data is not transposed first, when doing column reduction the memory access will be non-coalescent and require many more memory fetches than necessary. If, however, the B-scan is transposed, each row of the resulting matrix will be reduced to find the summation of the elements, employing coalesced memory access. Once implemented, we achieved better performance than the previous method. This is seen in Figure 4.4.

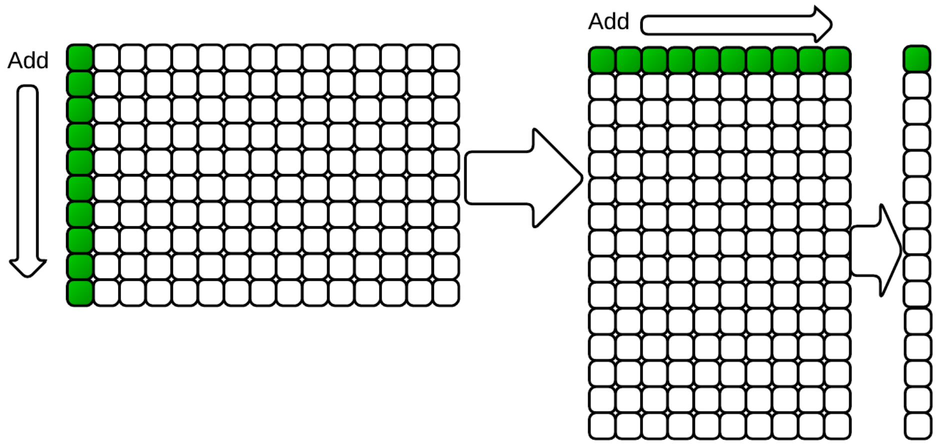


Fig. 4.4.: Transpose and reduce B-scan for DC subtraction.

We want to transpose the matrix first so that memory access will be coelesced. Similarly, the act of transposing a large matrix on a GPU takes some of its own tricks in order to run optimally. An excellent paper on this by Reutsch and Micikevicius [25] shows how to use tiling and shared memory as well as diagonal block reordering in order to get around these conflicts. Our transpose makes use of all of these techniques (tiling, etc.) to optimize our speed.

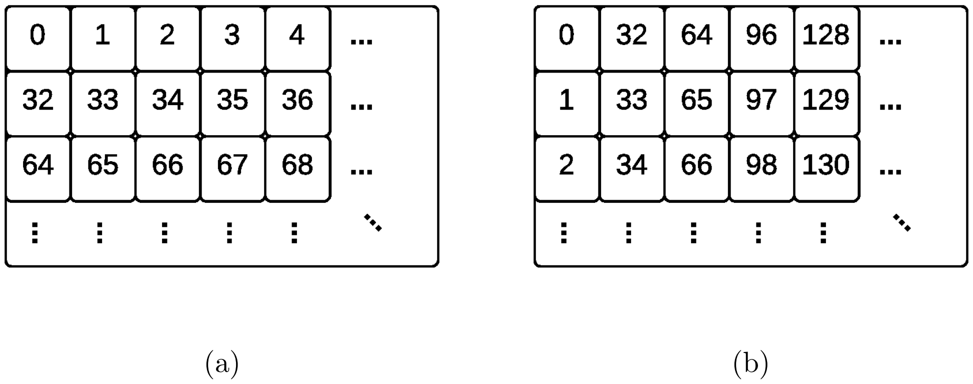


Fig. 4.5.: Transposing tile using shared memory: each block handles a tile of the matrix. A number represents a thread id, and a square an element in shared memory. (a) Each thread loads the data from the matrix row-wise, and (b) write the data back to global memory column-wise.

Once the B-scan is transposed, the reduction calculates the total along each row and stores the calculated average in a new vector. Algorithm 4.3 shows the procedure used for transposing the image, and Algorithm 4.4 shows the reduction procedure used to calculate the averaged A-line. After this step is complete, another kernel is launched that subtracts the average from each element as shown in Algorithm 4.5. All three steps are combined in Algorithm 4.2.

Algorithm 4.2 DC subtraction.

```
1: transpose⋘blocks,threads⋙                    ▷ Algorithm 4.3
2: getAverage⋘blocks,threads⋙                   ▷ Algorithm 4.4
3: subtractAverage⋘blocks,threads⋙              ▷ Algorithm 4.5
```

Algorithm 4.3 transpose⋘blocks,threads⋙ (in, out, width, height, TILE_DIM, BLOCK_ROWS)

Note: Variable *in* is the untransposed matrix, *out* is the transposed matrix, *width* and *height* are from the size of the B-scan, and *TILE_DIM* and *BLOCK_ROWS* are the tile dimensions.

```
1: bid ← blockIdx.x + gridDim.x × blockIdx.y
2: blockIdx_y ← bid  mod  gridDim.y
3: blockIdx_x ← ( ( bid / gridDim.y ) + blockIdx_y )  mod  gridDim.x
4: xIndex ← blockIdx_x × TILE_DIM + threadIdx.x
5: yIndex ← blockIdx_y × TILE_DIM + threadIdx.y
6: index_in ← xIndex + ( yIndex ) × width;
7: xIndexO ← blockIdx_y × TILE_DIM + threadIdx.x;
8: yIndexO ← blockIdx_x × TILE_DIM + threadIdx.y;
9: index_out ← xIndexO + ( yIndexO ) × height;
10: if xIndex < width then
11:     for all i such that 0 ≤i < TILE_DIM do
12:         if (yIndex + i) < height then
13:             tile[threadIdx.y + i][threadIdx.x] ← in[index_in + i × width];
14:         end if
15:         i ← i + BLOCK_ROWS
16:     end for
17: end if
18: __syncthreads()
19: if xIndexO < height  then
20:     for all i such that 0 ≤ i < TILE_DIM do
21:         if (yIndexO + i) < width then
22:             out[index_out + i × height] ← tile[threadIdx.x][threadIdx.y + i]
23:         end if
24:         i ← i + BLOCK_ROWS
25:     end for
26: end if
```

In Algorithm 4.3, the inputs to the algorithm are in, width, height, TILE_DIM, and BLOCK_ROWS. The variable in is the matrix that is being transposed. The variables width and height are from the size of the entire B-scan and are used to keep the threads in bounds. The variables TILE_DIM and BLOCK_ROWS are dimensions of the tile. The variable out is where the transposed matrix is stored. Lines 1 through 9 are calculating the index of the elements of the matrix that will be read into shared memory and the index of the elements of the resulting transposed matrix that will be written out. There are a couple of ideas being used in these calculations. First, the matrix is divided into tiles, submatrices, that each block works on. Second, blocks are ordered using something called diagonal block reordering. To learn more about these two concepts we would recommend the paper by Reutsch and Micikevicius [25]. In Lines 10 through 17, the kernel is reading in data from the matrix row-wise and storing that data in local shared memory. Line 18 synchronizes the threads, making sure that the entire tile is entirely read into memory. Since the number of threads in the block is larger than an individual warp it is very important to keep the threads synchronized, otherwise some threads could be at the writing out stage in the next steps, prior to when the data is read in, meaning that the kernel will write out garbage data before the desired data is available. Last, in Lines 19 through 26, the tile of data that was stored in shared memory, is written out to the transposed matrix. The writing is row-wise, preserving the coelesced memory access to global memory on the GPU.

Algorithm 4.4 getAverage$\lll$blocks,threads$\ggg$ (in, out, n)

Note: Variables *in* and *n* are the inputs. Variable *in* is the transposed matrix, and *n* is the width of the transposed matrix. Variable *out* is the resulting averaged A-line.

```
1:  tid ← threadIdx.x
2:  i ← blockIdx.x × blockSize + threadIdx.x
3:  gridSize = blockSize × gridDim.x
4:  mySum ← 0
5:  while  i < n do
6:      mySum ← mySum + in[i + blockIdx.y × n]
7:      i ← i + gridSize
8:  end while
9:  sharedData[tid] = mySum
10: __syncthreads()
11: b ← blockSize
12: while b > 64 do
13:     if tid < b / 2 then
14:         sharedData[tid] ← sharedData[tid] + sharedData[tid + b / 2]
15:         __syncthreads()
16:     end if
17:     b ← b / 2
18: end while
19: if tid < 32 then
20:     while b > 1 do
21:         sharedData[tid] ← sharedData[tid] + sharedData[tid + b / 2]
22:         b ← b / 2
23:     end while
24: end if
25: if tid = 0 then
26:     out[blockIdx.y] ← sharedData[0] / n
27: end if
```

In Algorithm 4.4 we are performing reduction on each row of the transposed matrix from Algorithm 4.3. The result will be an averaged A-line. In Lines 1 through 3, we are finding the thread information and the position of the row that we are averaging. Line 4 initializes a variable used to hold our initial summation and aide in loading the rows into shared memory. Lines 5 through 9 load the pixels in the row into local memory. The rows that we are averaging can be an arbitrary width (the width of this row is the height of the columns in the B-scan), and the shared local memory has a width of the highest power of two under the transposed matrix width. The pixels at indexes greater than the shared memory width are loaded by adding them into the shared memory. Line 10 synchronizes the threads. Lines 11 through 18 start the reduction. Each iteration through the while loop, the number of active threads is cut in half. After each addition, the threads are synchronized in Line 15. This is because the number of active threads is still greater than a warp and threads on different warps run at different times. Once the active threads are down to a warp the algorithm moves on to Lines 19 through 24 where synchronization is no longer needed. The threads continue to be cut into half until the summation is contained in the first element of the sharedData array. Finally, in Lines 25 through 27, the first thread of each block outputs the average calculated in that block.

Algorithm 4.5 subtractAverage⋘blocks,threads⋙ (in, out, avg, width)

Note: *in*,*avg*,*width* are the input variables. *in* is the B-scan from before the transpose was performed in Algorithm 4.3. *avg* contains the averaged A-line, and *width* contains the width from the size of the B-scan. *out* is the output variable where the resulting B-scan is written to.

```
1: idx ← blockDim.x × blockIdx.x + threadIdx.x
2: idy ← blockDim.y × blockIdx.y + threadIdx.y
3: if idx < width then
4:     out[idy × width + idx] ← in[idy × width + idx] − avg[idx]
5: end if
```

Algorithm 4.5 concludes Algorithm 4.2. Lines 1 and 2 find the column and row information, respectively, of the matrix (the matrix prior to the tranposition) as well as the avgeraged A-line calculated from Algorithm 4.4. In Lines 3 through 5 we subtract the averaged A-line value from the matrix and store the result in global memory.

4.3 Zero Padding

The next step, Line 4 of Algorithm 4.1, is padding A-lines. This is done for a couple of reasons. The foremost is because it produces a better resulting image, but also padding the A-lines out to a power of two is more efficient for the FFTs and general use on the GPU.

This process includes three steps: 1D-FFT of each A-line, padding the center of the resulting frequency information with zeros, and 1D-IFFT to return back from the frequency domain. This is shown in Algorithm 4.6 with the middle padding shown in Algorithm 4.7. By padding it this way, using FFTs, the signal is spread over the padded width. The data going into the FFT is real; due to this, the data after the FFT exhibits Hermitian symmetry. Detailed more in Section 4.4, the Hermitian symmetry allows the resulting data to be stored in $N/2+1$ complex values. The padding then occurs from $N/2+2$ until $N_{\text{padded}}/2+1$.

Algorithm 4.6 Pad

```
1: ComplexArray ← FFT real to complex
2: kernelPad⋘blocks,threads⋙ (ComplexArray)
3: RealArray ← IFFT complex to real
```

In Algorithm 4.6 we use cuFFT for Lines 1 and 3 and our own kernel for Line 2 described in Algorithm 4.7.

Algorithm 4.7 kernelPad⋘blocks,threads⋙ (ComplexArray, inWidth, outWidth, height)

Note: *ComplexArray* contains the output of the previous FFT. Variables *inWidth* and *outWidth* dictate what the width of the output of the FFT is and the input of the IFFT, respectively. *height* is the number of A-lines in a B-scan.

```
1: idx ← blockDim.x × blockIdx.x + threadIdx.x
2: idy ← blockDim.y × blockIdx.y + threadIdx.y
3: out ← idy × outWidth + idx
4: if idx ≥ inWidth & idx < outWidth then
5:     for all $i$ such that $0 \le i <$ height do
6:         ComplexArray[out] ← 0
7:         out ← out + outWidth
8:     end for
9: end if
```

Algorithm 4.7 sets the padded area, as shown in Figure 4.6, to zero. Lines 1 through 3 determine the indexes of the elements that are to be set to zero. Line 4 blocks any threads from going outside of that area. Lines 5 through 8 step through a column and sets the elements to zero.

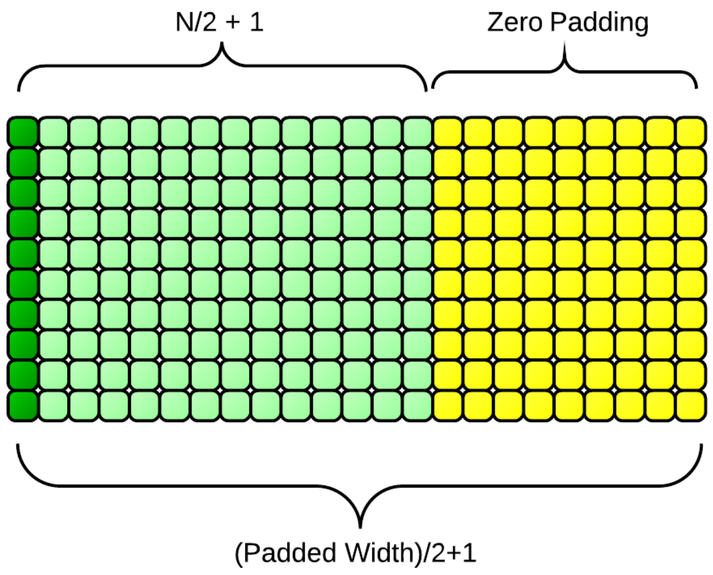


Fig. 4.6.: Storage of values when padding.

4.4 CuFFT

One of the strengths of CUDA in recent years has been the support of NVIDIA and other GPGPU programmers in developing parallel packages that are useful to a broad range of people. One such area is fast-Fourier transforms (FFTs). The cuFFT library is NVIDIA's FFT package for the GPU. It was based on FFTW, a widely used CPU FFT package [26]. What is actually being calculated by an FFT is the Discrete Fourier Transform (DFT) shown in Equation 2.3 and Equation 2.4 in the forward and reverse directions, respectively.

In processing AOOCT data, as explained in Sections 2.2 and 2.2.5, FFTs are used a total of three times for each A-line to transform it from raw data to a finished output image. A forward DFT and a reverse DFT are used in padding the input data in

Lines 1 and 3 of Algorithm 4.6, respectively, and another forward DFT is used upon completion of the calculations in Line 7 of Algorithm 4.1. While offering a litany of useful features, the particulars useful to our calculations are batching, complex and real valued input and output, strided layout, and streamed execution [26]. As seen in Table 4.1, the data varies in types for different operations through out the processing, and the ability to specify complex or real is an advantage. The batching is also necessary. Each FFT is performed on one A-line at a time. The ability to batch the FFT calls over the whole B-scan is necessary for performance.

Table 4.1: Data flow through processing.

Step	**Format**	**Byte per Element**	**Elements per Aline**
Raw Data	UInt16	2	Arbitrary
Pre Padding	Float	4	Arbitrary
Padding	Complex Float	8	Arbitrary
Post Padding	Real Float	4	4096
...	Real Float	4	4096
Post Disp. Comp.	Complex Float	8	4096
Post FFT	Complex Float	8	4096
Pre Display	UInt8	1	< 2048

The cuFFT package is very flexible in that it allows for specifying the input and output arrays to the FFT. By default, the FFT, whether operating in-place or going from an input to an output or whether going from complex to real or real to complex, uses space as seen in Table 4.2.

Table 4.2: CuFFT data sizes used in different operations.

FFT Type	**Input Data Size**	**Output Data Size**
Complex to complex	x	x
Complex to real	$\lfloor \frac{1}{2}x \rfloor + 1$	x
Real to complex	x	$\lfloor \frac{1}{2}x \rfloor + 1$

Instead of this default behavior, it is more desirable to keep the number of elements constant. An advanced layout mode enables a user to be precise in the number of input and output elements and how those are arranged for batched operations [26]. In full operation, we utilize three FFTs: a real-to-complex FFT, a complex-to-real inverse FFT, and a complex-to-complex final FFT. The library allows us to use complicated storage of data and batched processing. It also takes advantage of hermitian symmetry to save on storage space, and the reduced size makes the resulting actions more efficient and faster. For example, when going through padding, a 2048-wide A-line would be stored in 2048 floats. After the FFT, it would be 2048 complex values (4096 floats). After padding, it would be 4096 complex values (8192 floats). Then finally it would go through the inverse FFT to be 4096 real values due to symmetry (4096 floats).

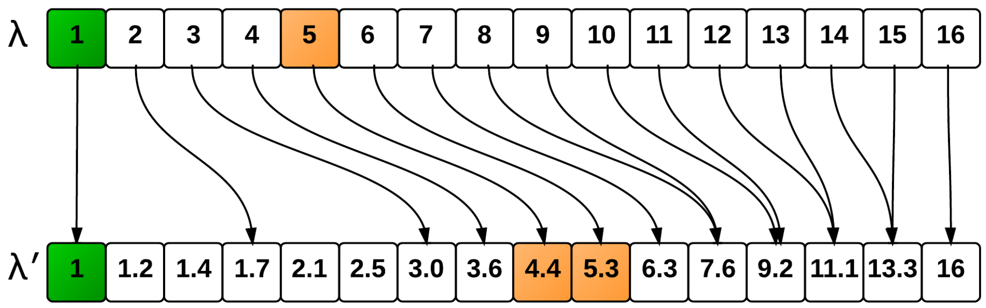
Because of the symmetry, cuFFT stores the symmetric array in $N/2+1$ elements. Thus, the 2048-wide A-line goes into 1025 complex values after the first FFT (2050 floats). After padding, the A-line is in 2049 complex values (4098 floats). Also, using advanced layout, the initial FFT places the 1025 complex value A-lines into an array with 2049 complex values per A-line as shown in Figure 4.6 Looking at the array on the B-scan level, the overall array is 2049 complex value wide by the number of A-lines per B-scan. As a matrix it would have 2049 columns, and the number of

A-lines per B-scan rows. The first 1025 complex value columns are occupied by the results of the FFT, and the remaining 1024 complex value columns must be zeroed out to complete the padding.

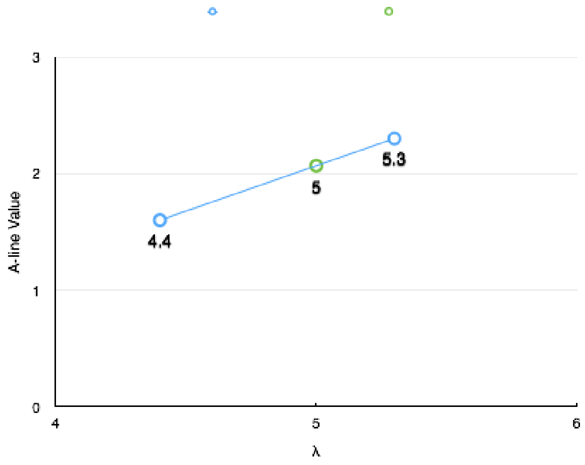
The cuFFT library relies on making a plan for an FFT. We utilize a separate plan for each type of FFT: real-to-complex forward FFT, complex-to-real inverse FFT, and a complex-to-complex forward FFT. When making a plan, the cuFFT library allocates global memory for its operations. While this may not be obvious to the user, it is important in this instance: since FFTs could be performed concurrently on separate streams, we need separate plans for each FFT [26].

4.5 K-Space Mapping

Line 5 of Algorithm 4.1 is to align the A-lines to k-space The reasoning for k-space mapping was explained in Section 2.2.3, as were the basic ideas. In practice, we use a simple linear interpolation to find the new values for the A-line spectra. This requires a two step process. First, the new values are mapped to old values, to find where the interpolation is going to take place. The mapping searches the old wavelengths to find the element that is just left or equal to the value being mapped as seen in Figure 4.7.



(a)



(b)

Fig. 4.7.: Mapping of new evenly-spaced wavelengths to configured wavelengths. Each pixel of the incoming A-lines is associated with a wavelength. In a cartesian system, one could think of the A-line values as y values and the wavelengths as x values. (a) New evenly spaced wavelengths are mapped to the old wavelengths that are provided from a configuration file. (b) To find the new value for the A-line at $\lambda' = 5$, the incoming A-line values from where $\lambda = 4.4$ and $\lambda = 5.3$ are used to perform a linear interpolation.

The first step is not an ideal parallel problem because the mapping of each element is somewhat independent. In essence, a search is being performed for each λ' to find the location in the λ array where $\lambda_i \leq \lambda' \leq \lambda_{i+1}$. Because some searches will end earlier than others, the threads that finish early must still wait until the other threads in the same warp finish, causing the cores to sit idle. The list of wavelengths is monotonically increasing, which allows better than O(N) search time, but each thread is possibly going to need different time to find the appropriate mapping. This method is shown in Algorithm 4.8. However, the good news is that every A-line in a scan will use the same wavelengths, so this mapping only needs to be done once. In the mapping kernel, a vector is created and stored in global memory that contains the location of the $\lambda_{\text{left}}i$ which is the value that is less than or equal to the new wavelength.

During the actual interpolation kernel, Equation 4.1 is performed to find the new A-line spectra as shown in Algorithm 4.9.

$$A'_i = A_{\text{left}i} + (A_{\text{right}i} - A_{\text{left}i}) \frac{\lambda'_i - \lambda_{\text{left}i}}{\lambda_{\text{right}i} - \lambda_{\text{left}i}} \tag{4.1}$$

where $A_{\text{left}i}$ and $A_{\text{right}i}$ are from the incoming A-line, and A'_i is the new value that we are interpolating from the incoming data. Only the interpolation described in Equation 4.1 is performed for each A-line streaming through the program.

Algorithm 4.8 findLefts⋘blocks,threads⋙ (configuredWavelengths, newWavelengths, width, mapping)

Note: *configuredWavelengths*, *newWavelengths*, and *width* are the input variables. *configuredWavelengths* and *newWavelengths* are the wavelengths determined during configuration and the new evenly-spaced in k-space wavelengths that we are interpolating the A-lines to, respectively. *width* is the size of an A-line. *mapping* is the output of this algorithm and is an array containing the mapping of *newWavelengths* to *configuredWavelengths* so that the interpolation in Algorithm 4.9 can be done.

```
1: idx ← blockDim.x × blockIdx.x + threadIdx.x
2: sx[idx] ← configuredWavelengths[idx]
3: __syncthreads()
4: test ← 0
5: left ← idx
6: x ← newWavelengths[idx]
7: max ← width − 1
8: min ← 0
9: repeat
10:     if left < 0 then
11:         left ← left + 1
12:     else if left + 1 ≥ width then
13:         left ← left − 1
14:     end if
15:     if sx[left + 1] < x then
16:         min ← left
17:         left ← (max − left) / 2 + left
18:     else if sx[left] > x then
19:         max ← left
20:         left ← (left − min) / 2 + min
```

```
21:     else
22:         test ← 1
23:     end if
24: until test = 0
25: mapping[idx] ← left
```

Lines 1 through 3 of Algrorithm 4.8 set the index of the wavelengths being looked at and load the configured wavelengths into shared memory. Line 4 initializes a test variable that will be flipped to 1 once the correct mapping is complete. Line 5 initializes the left variable. Line 6 loads the newWavelength being examined by a thread into a local variable x. Lines 7 and 8 initialize the max and min index search areas. In Lines 9 through 24 the search area is iteratively cut down until an index is found for variable left where sx[left] l x $\leq$ sx[left+1]. Lines 10 through 14 act as a boundary. Lines 15 through 22 test to see if the condition just mentioned is met, and if not then the search area is cut down. If the current value of left produces an sx[left] that is to large then the max is moved to that value, otherwise, if the value of left produces an sx[left+1] that is too small, then the min is moved to that value. Line 25 saves the result out to variable mapping on global memory.

Algorithm 4.9 kInterpolation⋘blocks,threads⋙ (xIn, yIn, xOut, yOut, mapping, width,n)

Note: *xIn* and *yIn* are input arrays containing the wavelengths and A-line data, respectively, from the previous padding step in Algorithm 4.6. *mapping* contains the mapping from Algorithm 4.8. *width* contains the width and *n* contains the height of the B-scan. *xOut* is also an input, it contains the newWavelengths that the interpolated A-lines will be associated with. *yOut* is the output of this algorithm, containing the interpolated A-lines.

```
1: idx ← blockDim.x × blockIdx.x + threadIdx.x
2: idy ← blockDim.y × blockIdx.y + threadIdx.y
3: left ← mapping[idx]
4: x ← xOut[idx]
5: xLeft ← xIn[left]
6: xRight ← xIn[left + 1]
7: for all $i$ such that $0 \le i < n$ do
8:     in ← (idy + i) × width + left
9:     out ← (idy + i) × width + idx
10:    m ← ( yIn[in + 1] − yIn[in] ) / ( xRight − xLeft)
11:    yOut[out] ← m × (x − xLeft) + yIn[in]
12: end for
```

Lines 1 through 6 identify the indexes required to reference the configured wavelengths and incoming A-lines and the new wavelengths. They also load those values into memory. Lines 7 through 11 perform the linear interpolation calculation.

4.6 Dispersion Compensation

This step of processing removes an optical effect called dispersion, common to OCT images, that blurs the image. To remove the blur, a calibration file is provided that contains complex phase terms. Each pixel in an A-line is multiplied by a complex phase term. In terms of CUDA, this is a very simple kernel that reads in B-scans and multiplies by the complex coefficients provided from calibration as shown in Algorithm 4.10.

Algorithm 4.10 compensateForDispersion⋘blocks,threads⋙ (phaseCorrection, RealIn, ComplexOut, width, height)

Note: *phaseCorrection* is an array of complex coefficients. *RealIn* is the array of real-valued A-line data after it has been realigned to k-space in Algorithm 4.9. *width* and *height* are from the size of the B-scan. *ComplexOut* is the complex valued A-lines output from this algorithm. For complex values *.Real* references the real value and *.Imag* references the imaginary value.

```
1: idx ← blockDim.x × blockIdx.x + threadIdx.x
2: idy ← blockDim.y × blockIdx.y + threadIdx.y
3: gid ← idy × width + idx
4: pC ← phaseCorrection[idx]
5: for all $i$ such that $0 \leq i <$ height do
6:     ComplexOut[gid] ← pC.Real × RealIn[gid] + pC.Imag × RealIn[gid]
7:     gid ← gid + width
8: end for
```

Lines 1 through 3 calculate the indexes to reference the input and output arrays. Line 4 loads the dispersion coefficient into memory. Lines 5 through 8 perform the complex multiplication and save the result to global memory.

4.7 Streaming

The Fermi architecture and Compute 2.0+ can simultaneously support:

- Up to 16 GPU kernels.
- Two asynchronous memory copies (provided they are in different directions)
- CPU computation.

Each of GPU kernels will be launched on a separate stream. With the Kepler architecture and Compute 3.5, devices can support up to 32 streams, therefore up to 32 concurrent kernels [23]. While the device can support up to 16 or 32 streams, increasing the streams may or may not increase throughput. There exists a finite amount of resources on the device that can be computing at any given time. So while streams are able to be concurrent, if the resources are already being used, a stream will be scheduled and have to wait until resources become available.

Using streams does produce noticeable results, and is integral to keeping up with the incoming data. The streams operate in a pipelined fashion, as seen in Figure 4.8. Figure 4.9a and 4.9b show how the streams look in time using NVIDIA's Visual Profiler. Each block in a row is time that a kernel is running.

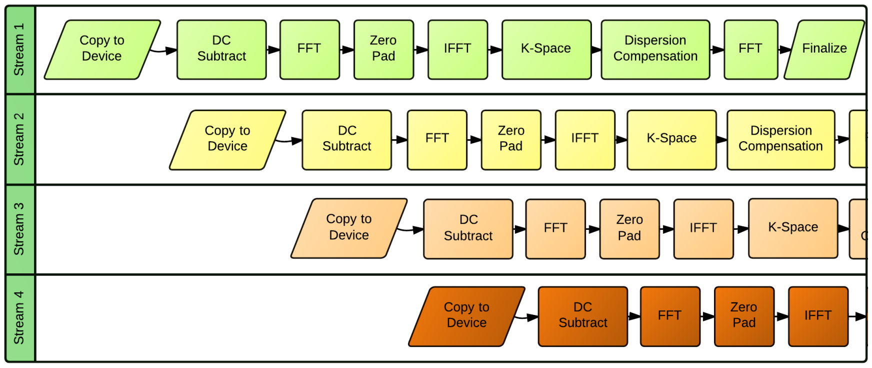
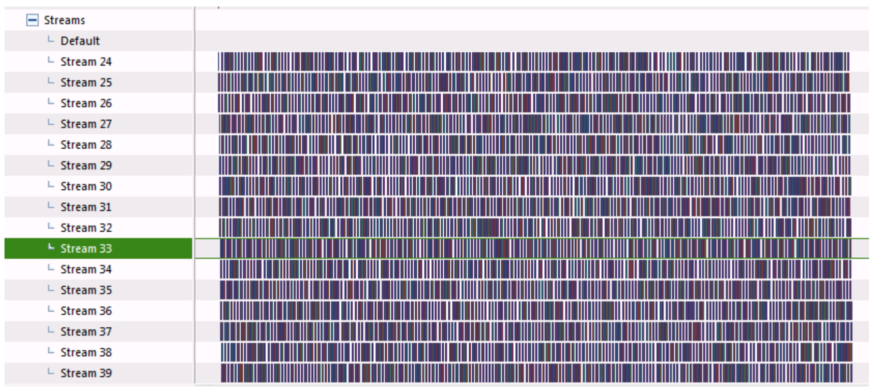
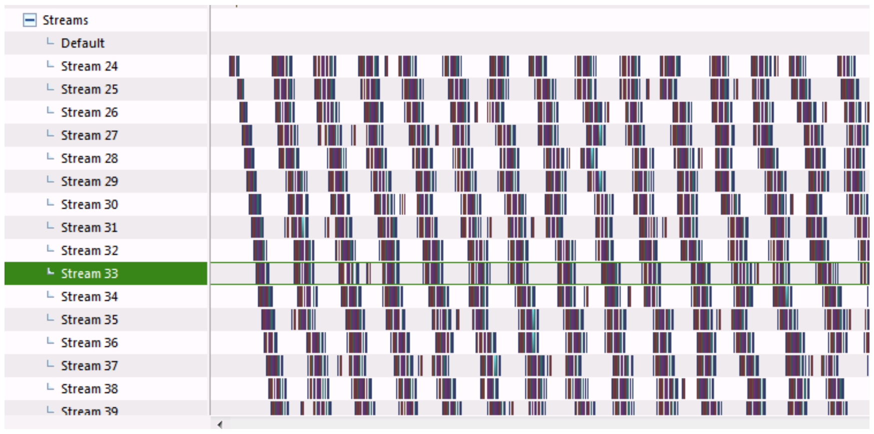


Fig. 4.8.: Pipelining of image processing.



(a)



(b)

Fig. 4.9.: NVIDIA Visual Profiler timeline image of streams where (a) is over a wider slice of time than (b), which is over a narrower, smaller subset of time.

One of the most notable benefits of using streams in this application is hiding the latency of copying data from CPU RAM to the GPU RAM. To move data from the CPU and more specifically from the RAM to the GPU RAM, the data streams via the PCIe bus, the standard interface that graphics and other cards are connected to the motherboard in a computer. This takes many clock cycles to accomplish, which means there are a lot of instructions that can be carried out on CUDA cores during the time that the data is transferred. Over PCIe 2.0, there is an effective cap at 500MB/s per lane. The Tesla K20c can use up to 16 lanes, giving a maximum of

8GBs. Whether the throughput of the PCIe bus is high or not, streams allow the GPU to continue computation while data is transferred asynchronously. The GPU is capable of doing two memory transfers (in and out) concurrently with computation. This is invaluable in hiding the memory copy latency.

In order to facilitate streams, it is advantageous to give each stream an exclusive, clear path for the data to flow through, independent of the other streams. If the streams used the same memory, it would be necessary to synchronize across streams to prevent corruption of the data. While possible, this would be an added complication and could lower the throughput. Fortunately, in this application, there is enough global memory on the GPU to create separate paths for each stream. Each stream uses its own designated memory for each step of the process as seen in Figure 4.10. Also, it is important to be aware of memory usage by libraries like cuFFT. CuFFT requires a plan to be made for the type and parameters of the FFT desired prior to execution. The plan is a configuration for cuFFT in what algorithm it will use and how the data will be moved and is necessary for execution. CuFFT provides built-in functions that generate a plan using parameters of the FFT. Then when one wants to execute the FFT, the execution references the plan that was made previously. While not explicit in the documentation, each time a plan is generated, global memory is allocated to perform the FFT. Therefore, a plan is needed for each stream as well.

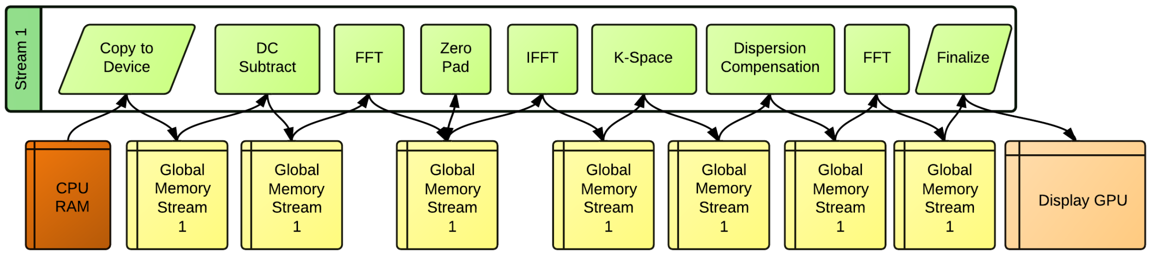


Fig. 4.10.: Use of global memory with a CUDA Stream.

5. RESULTS

To compare the performance of AOOCT on a GPU against the performance on a CPU, the equivalent operations on a CPU were written in C/C++. Since cuFFT was originally developed from FFTW, FFTW was used to perform the FFTs in the CPU version. All data is handled in single precision. The performance times are recorded in Table 5.1. Using the GPU version on a Tesla K20c provides at least a 34$\times$ speedup over the Intel Xeon, and 75$\times$ speedup over the Intel Core i7. This is computed over a data set consisting of 11 volumes containing the characteristics in Table 5.2. The initial times used by Dr. Miller's research group at Indiana University-Bloomington using MATLAB were over 47 *minutes*. The GPU exhibits over 1900$\times$ speedup as compared to this implementation.

Table 5.1: AOOCT Runtimes on GPU vs CPU

Device	**B-scan**	**Volume μ^***	**Volume σ^{**}**
Intel Xeon	18.95ms	4549ms	96.18ms
Intel Core i7	41.4ms	9925ms	805.7ms
NVIDIA Tesla K20c	0.549ms	131.74ms	0.468ms

* Time average over 30 runs.

**Standard deviation of total volume time.

As for accuracy, we have compared the linear normalized image output from our CUDA implementation against the same steps performed using matlab. Pixel values ranged from 0 to 255, and the largest difference between the two implementations was observed to have a magnitude of 1. The matlab calculations are done using double

Table 5.2: AOOCT Data Parameters

Parameter	**Value**
A-line Length	832
A-lines per B-scan	240
B-scans per Volume	240
Volumes	11

precision compared to single precision on our GPU implementation, and mathematical calculations on the GPU can have slightly different rounding errors than on the CPU. With those known differences a variation of +/- 1 per pixel is acceptable.

5.1 Future Work

The overarching goal of this work is to make AOOCT a tool available in the clinical environment. To that end, this work enables the processing to be done in real time, but there is still much work to be done to make a unified platform commercially available. In the realm of the computation, image registration and segmentation are low hanging fruit for the near term. One of the issues with imaging *in vivo* is movement. There exists a lot of movement in the eye during a scan, which translates into image volumes that jump from B-scan to B-scan. This problem can be alleviated with image registration and correction from B-scan to B-scan.

5.2 Conclusion

By using CUDA-C, pipelining data into data streams, optimizing individual kernels, and optimizing the load of resources for different kernels in different steps of the process, we have been able to provide a real-time implementation of AOOCT image processing. As of this writing, our work is in the final stages of being implemented by Dr. Miller's research group in Bloomington, and will provide a basis for advances in the near future.

LIST OF REFERENCES

LIST OF REFERENCES

[1] J. Porter, A. Awwal, J. E. Lin, H. M. Queener, and K. Thorn, eds., *Adaptive Optics for Vision Science: Principles, practices, design and applications*. Wiley Series In Microwave and Optical Engineering, Wiley-Interscience, 2006.

[2] A. Pallikaris, D. R. Williams, and H. Hofer, "The reflectance of single cones in the living human eye," *Investigative Ophthalmology & Visual Science*, vol. 44, no. 10, pp. 4580–4592, 2003.

[3] J. Rha, R. Jonnal, Y. Zhang, and D. Miller, "Rapid fluctuation in the reflectance of single cones and its dependence on photopigment bleaching," *ARVO Meeting Abstracts*, vol. 46, no. 5, p. 3546, 2005.

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

[5] 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 Surg Lasers Imaging Retina*, vol. 42, pp. S56–S66, 2011.

[6] 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.

[7] R. A. Goldberg, S. P. Shah, and J. S. Duker, "Optical coherence tomography in macular hole management," *Retinal Physician*, vol. 10, April 2013.

[8] A. Maalej, W. Cheima, K. Asma, R. Riadh, and G. Salem, "Optical coherence tomography for diabetic macular edema: Early diagnosis, classification and quantitative assessment," *Journal of Clinical & Experimental Ophthalmology*, 2012.

[9] K. Nouri-Mahdavi, K. Nikkhou, D. C. Hoffman, S. K. Law, and J. Caprioli, "Detection of early glaucoma with optical coherence tomography (StratusOCT)," *Journal of Glaucoma*, vol. 17(3), pp. 183–188, April/May 2008.

[10] Y. Jian, K. Wong, and M. V. Sarunic, "Graphics processing unit accelerated optical coherence tomography processing at megahertz axial scan rate and high resolution video rate volumetric rendering," *Journal of Biomedical Optics*, vol. 18, no. 2, pp. 026002–026002, 2013.

[11] K. Zhang and J. U. Kang, "Graphics processing unit-based ultrahigh speed real-time fourier domain optical coherence tomography," *Selected Topics in Quantum Electronics, IEEE Journal of*, vol. 18, pp. 1270–1279, July 2012.

[12] R. J. Zawadzki, Y. Zhang, S. M. Jones, R. D. Ferguson, S. S. Choi, B. Cense, J. W. Evans, D. Chen, D. T. Miller, S. S. Olivier, and J. S. Werner, "Ultrahigh-resolution adaptive optics - optical coherence tomography: toward isotropic 3μm resolution for in vivo retinal imaging," 2007.

[13] D. T. Miller, O. P. Kocaoglu, Q. Wang, and S. Lee, "Adaptive optics and the eye (super resolution OCT)," *Eye*, vol. 25, pp. 321–330, Mar 2011.

[14] 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, and J. G. Fujimoto, "Optical coherence tomography," *Science*, vol. 254, p. 1178, Nov 22 1991. Copyright - Copyright American Association for the Advancement of Science Nov 22, 1991; Last updated - 2010-06-08; CODEN - SCIEAS.

[15] A. F. Fercher, C. K. Hitzenberger, W. Drexler, G. Kamp, and H. Sattmann, "In vivo optical coherence tomography," *American journal of ophthalmology*, vol. 116, no. 1, pp. 113–114, 1993.

[16] E. A. Swanson, J. A. Izatt, M. R. Hee, D. Huang, C. P. Lin, J. S. Schuman, C. A. Puliafito, and J. G. Fujimoto, "In vivo retinal imaging by optical coherence tomography," *Opt. Lett.*, vol. 18, pp. 1864–1866, Nov 1993.

[17] Y. Zhang, J. Rha, R. Jonnal, and D. Miller, "Adaptive optics parallel spectral domain optical coherence tomography for imaging the living retina," *Opt. Express*, vol. 13, pp. 4792–4811, Jun 2005.

[18] B. Cense, N. Nassif, T. Chen, M. Pierce, S.-H. Yun, B. Park, B. Bouma, G. Tearney, and J. de Boer, "Ultrahigh-resolution high-speed retinal imaging using spectral-domain optical coherence tomography," *Opt. Express*, vol. 12, pp. 2435–2447, May 2004.

[19] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Application of GPU Computing Series, Morgan Kaufmann, 1 ed., 11 2012.

[20] "EVGA GeForce GTX TITAN," 2013. `http://www.evga.com/Products/Product.aspx?pn=06G-P4-2790-KR` Last Date Accessed: Aug/2013.

[21] P. N. Glaskowsky, "NVIDIA's Fermi: The first complete GPU computing architecture," tech. rep., NVIDIA Corporation, 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf` Last Date Accessed: Oct/2013.

[22] "NVIDIA's next generation CUDA compute architecture: Kepler GK110," tech. rep., NVIDIA Corporation. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf` Last Date Accessed: Sept/2013.

[23] NVIDIA, "CUDA C programming guide," October 2012. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf` Last Date Accessed: Oct/2013.

[24] M. Harris, "Optimizing parallel reduction in CUDA." `http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf` Last Date Accessed: Oct/2013.

[25] G. Ruetsh and P. Micikevicius, "Optimizing matrix transpose in CUDA," January 2009. `http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf` Last Date Accessed: Oct/2013.

[26] "CUDA toolkit documentation: CUFFT," August 2013. `http://docs.nvidia.com/cuda/cufft/index.html` Last Date Accessed: Sept/2013.