Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Jeffery Edward Kriske Jr

Entitled
A SCALABLE APPROACH TO PROCESSING ADAPTIVE OPTICS OPTICAL COHERENCE TOMOGRAPHY DATA FROM MULTIPLE SENSORS USING MULTIPLE GRAPHICS PROCESSING UNITS

For the degree of Master of Science

Is approved by the final examining committee:

Fengguang Song

Jaehwan Lee

Rajeev Raje

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): Fengguang Song

Approved by: Shiaofen Fang 11/24/2014

Head of the Department Graduate Program Date

A SCALABLE APPROACH TO PROCESSING ADAPTIVE OPTICS OPTICAL COHERENCE TOMOGRAPHY DATA FROM MULTIPLE SENSORS USING MULTIPLE GRAPHICS PROCESSING UNITS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Jeffery E. Kriske Jr.

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

December 2014

Purdue University

Indianapolis, Indiana

ACKNOWLEDGMENTS

I would like to thank Dr. John Lee, Dr. Fengguang Song, Dr. Don Miller, and Dr. Rajeev Raje for their support, without which I would have been unable to continue the degree program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SYMBOLS

k	spacial frequency
λ	wavelength
π	pi
θ	phase

ABBREVIATIONS

AO	Adaptive Optics
AO-OCT	Adaptive Optics Optical Coherence Tomography
CMOS	Complementary MetalOxide Semiconductor
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FFT	Fast Fourier Transform
FLOPS	Floating Point Operations Per Second
GPGPU	General-Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
dB	decibels
DC	Direct Current
iFFT	Inverse Fast Fourier Transform
IU	Indiana University
OCT	Optical Coherence Tomography
RPE	Retinal Pigment Epithelium
SMX	Streaming Multiprocessor (Kepler Architecture)

NOMENCLATURE

A-scan	amplitude scan
B-scan	brightness scan
C-scan	*en-face* scan
en-face	front facing view
kernel	a multithreaded task running on the GPU
block	a group of parallel concurrently executing threads
grid	a group of thread blocks
warp	a group of 32 threads launched at the same time by an SMX

ABSTRACT

Kriske Jr., Jeffery E. MS, Purdue University, December 2014. A Scalable Approach to Processing Adaptive Optics Optical Coherence Tomography Data From Multiple Sensors Using Multiple Graphics Processing Units. Major Professors: John Jaehwan Lee and Fengguang Song.

Adaptive optics-optical coherence tomography (AO-OCT) is a non-invasive method of imaging the human retina *in vivo*. It can be used to visualize microscopic structures, making it incredibly useful for the early detection and diagnosis of retinal disease. The research group at Indiana University has a novel multi-camera AO-OCT system capable of 1 MHz acquisition rates. Until this point, a method has not existed to process data from such a novel system quickly and accurately enough on a CPU, a GPU, or one that can scale to multiple GPUs automatically in an efficient manner. This is a barrier to using a MHz AO-OCT system in a clinical environment. A novel approach to processing AO-OCT data from the unique multi-camera optics system is tested on multiple graphics processing units (GPUs) in parallel with one, two, and four camera combinations. The design and results demonstrate a scalable, reusable, extensible method of computing AO-OCT output. This approach can either achieve real time results with an AO-OCT system capable of 1 MHz acquisition rates or be scaled to a higher accuracy mode with a fast Fourier transform of 16,384 complex values.

1 INTRODUCTION

Our eyes are our windows to the world, without which, we cannot see. It is therefore important to protect our eyes and detect diseases as soon as possible. Optical coherence tomography (OCT) makes this possible *in vivo* because it can be used to visualize the microscopic structures of the retina. It is possible to detect potential problems even before the patient becomes symptomatic. OCT is an invaluable tool in a clinical setting to screen for retinal issues.

One of the difficulties with OCT is that the output must be computed to form a viewable image. The steps involved are computationally intensive, and producing a high accuracy version is even more so. While this computation can be done on any standard central processing unit (CPU), the computation time is prohibitive for a clinical environment even using mutlithreaded CPU-based software [1–3]. Fortunately, however, many of the computations can be done in parallel, which means that the more cores that work on the problem, the higher the throughput, depending on the acquisition speed. Although utilizing a cluster of computers or a supercomputer is an option, the latency incurred upon sending the data to a high performance computing cluster outside of a clinician's office would prevent real-time results.

The team at Indiana University (IU) Bloomington has a novel adaptive optics OCT (AO-OCT) system that requires a custom approach to processing the acquired data because the data is captured by multiple sources and interlaced to form the desired output at an increased acquisition speed. In addition, the custom solution should be scalable, extensible, and reusable.

A modern graphics processing unit (GPU) can provide thousands of processing cores, each of which can do work. If a task can be properly split up, it can benefit immensely from the additional computing power the GPU provides. However, the number of cores is not infinite and the workload can quickly saturate a GPU. In an

effort to circumvent this limitation, in this study an investigation into whether it is possible to scale a solution to multiple GPUs will also be conducted.

Many groups have processed standard OCT data on a single GPU [1, 2, 4, 5]. While processing AO-OCT data on multiple GPUs has been described [2, 6], it was known to cause instability unless different tasks are assigned on each GPU [6]. Yet, processing data captured by multiple sources in an interlaced fashion has never been studied before or in combination with scaling to process on multiple GPUs. One of the challenges involves keeping track of which camera each piece of data comes from in order to properly process that data. This thesis aims to describe such an approach while utilizing general processing on a graphics processing unit (dubbed GPGPU).

2 ADAPTIVE-OPTICS OPTICAL COHERENCE TOMOGRAPHY

Adaptive-optics optical coherence tomography (AO-OCT) is the combination of two technologies to allow a non-invasive view of a retina at micron-resolution. Micron resolutions allow for visualization at the cellular level. This can provide an invaluable resource for the early diagnosis of retinal diseases and in the study of animal models. The two technologies are optical coherence tomography (OCT) and adaptive optics (AO).

2.1 Optical Coherence Tomography and AO-OCT

Optical coherence tomography has been around for over two decades. It uses low-coherence interferometry to produce optical scattering data from microstructures within tissue. The early prototype time domain OCT (TD-OCT) produced images with an axial resolution of 15 μm [7]. Later the technology for TD-OCT was advanced to allow axial resolutions of 8-10 μm. A more advanced type of OCT is called spectral domain OCT (SD-OCT), which allows axial resolutions of 3 μm and faster capture times. SD-OCT can also resolve structures not visible with TD-OCT, which is useful when looking for disease states [8].

One of the problems with SD-OCT is that it is limited by aberrations on the ocular lens and cornea. To correct for these aberrations, adaptive optics can be added to an SD-OCT system. An AO-OCT system uses deformable mirrors to correct for aberrations in the optical path. It can also increase the transverse resolution to 1 μm [8].

An axial scan through tissue provides depth information at a specific point. This type of scan is called an A-scan, which is short for amplitude scan. Multiple A-scans are gathered linearly and produce a brightness scan, or B-scan. A B-scan can provide a cross-sectional view. When a volume formed from parallel B-scans is projected, it will form an *en-face* view of the retina. This orientation is shown in Figure 2.1

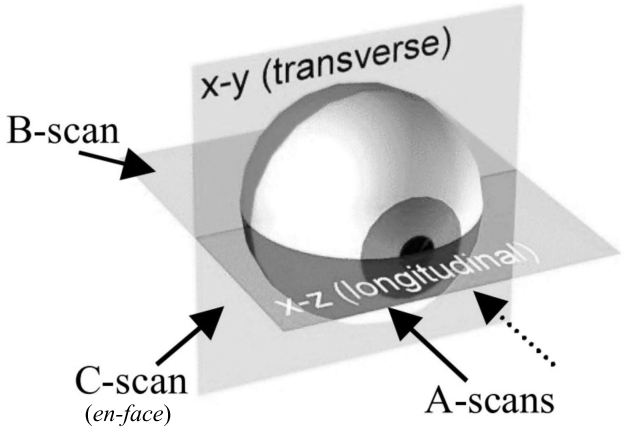

Figure 2.1. Relative scan orientation of A-scans, B-scans, and *en-face* views [9].

2.2 MHz AO-OCT Hardware

While the hardware of the IU Bloomington AO-OCT system follows the same basic concepts of any other AO-OCT system, what sets it apart is a unique multi-camera system. Instead of using a single camera set-up, the system uses four complementary metaloxide semiconductor (CMOS) line-scan cameras acting as four spectrometers. Each of these sensors by themselves are capable of an imaging acquisition rate of 250 KHz. Combined with a 1×4 optical switch assembly, these sensors together can achieve an imaging rate of 1 million A-scans per second (1 MHz). A sensor normally has a dead-time during which it cannot capture but reads out the data. During

this time other sensors can be synchronized to capture, and thus hiding this dead-time [10]. The resulting data stream contains interlaced A-scans from each of the sensors. Capturing A-scans at 1 million A-scans per second allows both a greater field of view and the ability to help reduce motion artifacts caused by motion of the eye during a retinal scan, thereby yielding a greater clinical value [10]. Until this point there has not been a real-time GPU-based approach at handling 1 MHz A-scan acquisition rates.

2.3 MHz AO-OCT Processing Steps

Data obtained by the AO-OCT system is not readily in a format that can be interpreted by a human being. Similar to an ultrasound, the data obtained must be processed in order to form a usable image. Processing this data is computationally intensive and even more so with higher levels of interpolation. The method used to prepare A-scans for processing is dependent on the implementation.

The steps to processing AO-OCT are as follows:

- Apply a Hann filter to taper the ends of short-length A-scans under 832 pixels
- Subtract the DC component
- Apply a Fourier transform
- Zero-pad the data
- Apply an inverse Fourier transform
- Map to k-space
- Compensate for dispersion
- Reconstruct the image using a Fourier transform
- Crop the data

- Convert the data to intensity values
- Normalize the data for both linear and log outputs
- Perform retinal tracking (optional)

Note that two main challenges exist here. First, DC subtraction, mapping to k-space, and dispersion compensation algorithms had to be designed so that it can work with one, two, or four cameras. They were created to solve the problem of having interleaved A-scans from multiple cameras, but still function correctly when A-scans come from a single camera.

Second, the challenge of processing B-scans is that it must be done as close to real-time as possible. If a clinician is attempting to focus on part of a retina, results need to be displayed on screen as adjustments are made to ensure focus.

2.3.1 Hann Filter

Due to physical limitations of the AO-OCT system, when spectrum lengths are shorter than 832 pixels, the edges of the spectrum will not reach zero. It is therefore necessary to apply a window function to the spectrum. A window function is also known as a tapering function because it will smooth the edges of the sample down to zero. The Hann function was chosen for this task [11]. The Hann function is applied to the first segment and the last segment of the spectrum sample at a user-defined range. The equation for each value is as follows:

$$w[n] = \frac{1}{2}(1 - \cos\frac{2\pi n}{N-1}) \tag{2.1}$$

where n is the current value, N is the width of the A-scan, and $w[n]$ is the resulting tapered value. The tapered values are then divided by the maximum value from the same segment. The resulting A-scan has ends that taper to zero. The process is shown in Figure 2.2.

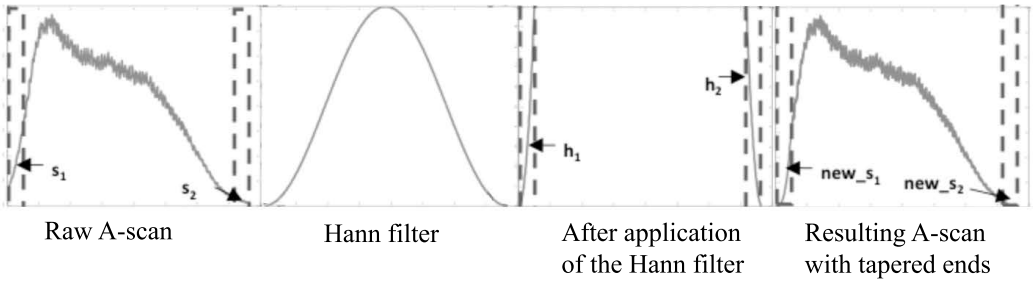

Figure 2.2. The Hann filter. The filter is applied to sections of the raw A-scan labeled s1 and s2. This forms h1 and h2. These sections are then normalized to the local maximum value resulting in the lines labeled new_s1 and new_s2.

2.3.2 Subtract Direct Current Component

One of the first steps in processing AO-OCT data is to subtract the direct current (DC) bias. This bias is present in the A-scan waveform and can be different between cameras and can even be different in the same camera between B-scans. In order to compensate for this, the DC bias is first calculated and then subtracted from each A-scan [12]. This process is shown in Figure 2.3

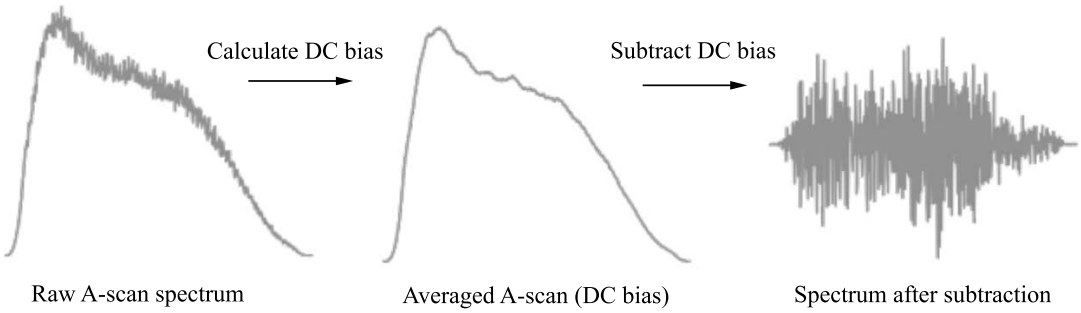

Figure 2.3. A general overview of the DC subtraction process.

To calculate and remove the DC bias for a B-scan, an average of all the A-scans from an individual camera must be subtracted from those same A-scans. This must

be repeated for the A-scans obtained from each of the cameras and the entire process is repeated on the next B-scan. This can be represented by:

$$A_n = a_n - \frac{1}{M}\sum_{k=0}^{M-1} a_{nk} \qquad n = 0, 1, 2, ..., N-1 \tag{2.2}$$

where M is the number of A-scans per camera per B-scan, and a_{nk} is the value of the n-th position on the k-th A-scan.

2.3.3 Zero Padding and Fourier Transforms

The spectra obtained from the OCT system need to be linearly interpolated into k-space, but linear interpolation is not inherently accurate. In order to improve accuracy of the interpolation, it is necessary to up-sample the data two to four times before interpolation [12].

In order to up-sample the spectra, first, a one-dimensional fast Fourier transform (FFT) is performed on each A-scan. A-scans are then zero-padded to a length of 4096. An inverse FFT (iFFT) is then performed on the resulting data, giving up-sampled data ready to be mapped to k-space or spacial frequency space. Up-sampling is a standard practice in digital signal processing, and the procedure is illustrated in Figure 2.4.

2.3.4 k-space Mapping

The term k-space refers to an array of numbers that represent spacial frequencies. In order to compensate later for dispersion of the spectral components and perform a reconstructive fast Fourier transform step, the spectra needs to be transformed from wavelength (λ) space into spacial frequency (k) space with evenly spaced k values.

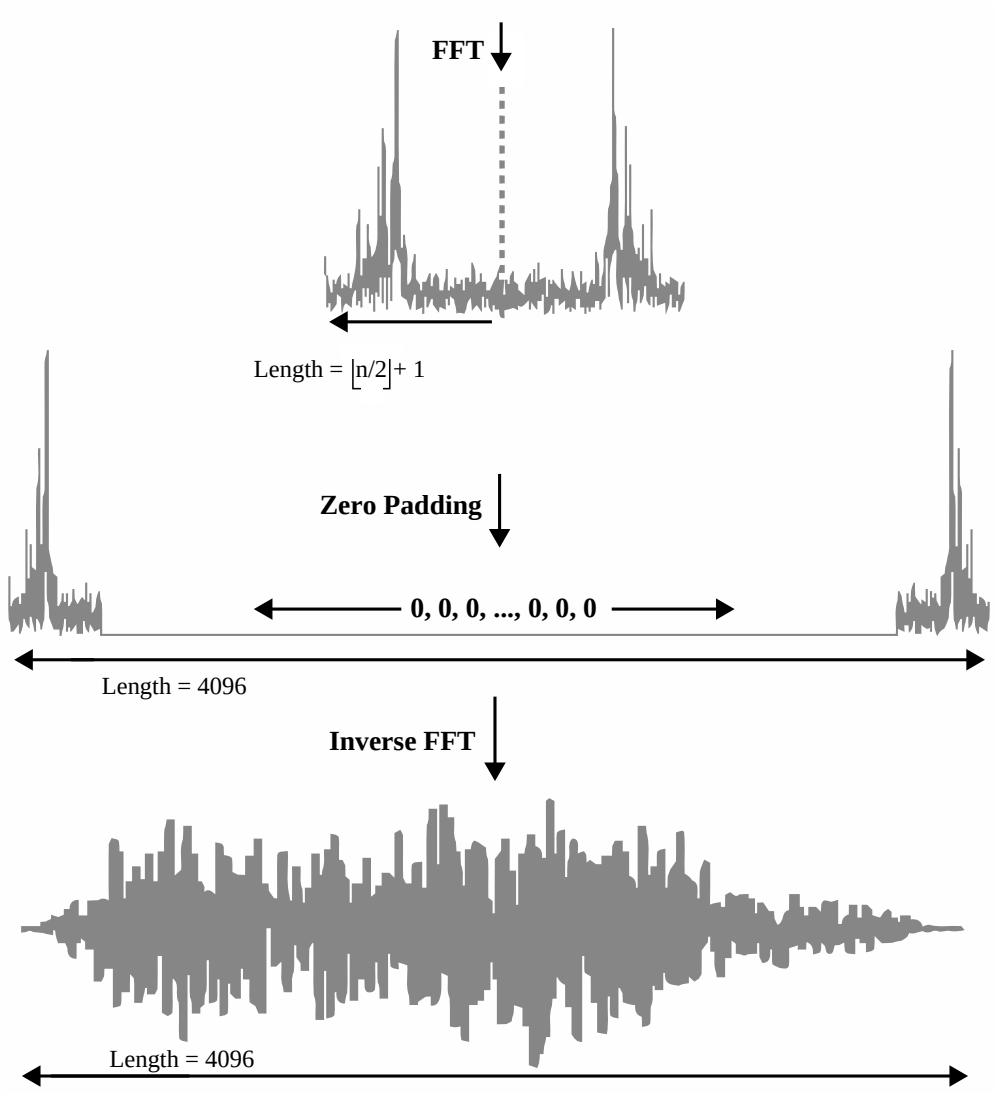

Figure 2.4. Up-sampling A-scans to a length of 4096.

In order to obtain linearly spaced k values, the maximum and minimum wavelengths of the spectrometer output must be used to create an array with the length of the original A-scan and divided evenly as follows:

$$\lambda_i = \lambda_{min} + i\left(\frac{\lambda_{max} - \lambda_{min}}{N-1}\right) \qquad i = 0, 1, 2, ..., N-1 \tag{2.3}$$

An interval is determined when $\left(\frac{\lambda_{max}-\lambda_{min}}{N-1}\right)$ is multiplied by i from zero to A-scan length minus one $(N-1)$. This is added to the minimum wavelength (λ_{min}).

Once an evenly spaced wavelength array is created, it is used to determine k values using the following equation:

$$k_i = \frac{2\pi}{\lambda_i} \tag{2.4}$$

With the k value array created, it is possible to use linear interpolation to map the previously up-sampled spectra to linearly sampled points in k-space [12].

2.3.5 Dispersion Compensation

The speed of light differs in materials that are optically dense or sparse. The more optically dense the material is, the slower the speed of light. In addition, the speeds of spectral wavelengths are affected differently while traveling through the medium, which can result in chromatic dispersion of the light. This effect increases linearly with the length of the material. When an eye with an unknown axial length is introduced to the OCT system, the tissue will cause chromatic dispersion, which should be compensated for in software [13].

In order to compensate for chromatic dispersion, a reference is taken during the calibration of the machine, and complex phase correction terms are calculated. These complex values are then multiplied to the k-space values determined from the previous step, resulting in a correction of chromatic dispersion. This step can be summarized as:

$$\theta_i = k_i \times \theta'_{col} \tag{2.5}$$

where θ'_{col} represents the complex phase correction values previously computed by the AO-OCT system at the column belonging to i, k_i represents the result from the previous step, and θ_i represents the resulting phase corrected value. The column is calculated using the formula $col = i \mod width$.

2.3.6 Reconstruction by Fourier Transform

Before the reconstruction happens, A-scans are again zero padded. Each scan is zero padded to four times the current size of 4096, resulting in A-scans with a length of 16, 384 complex values as shown in Figure 2.5 in order to increase the pixel resolution in the image domain.

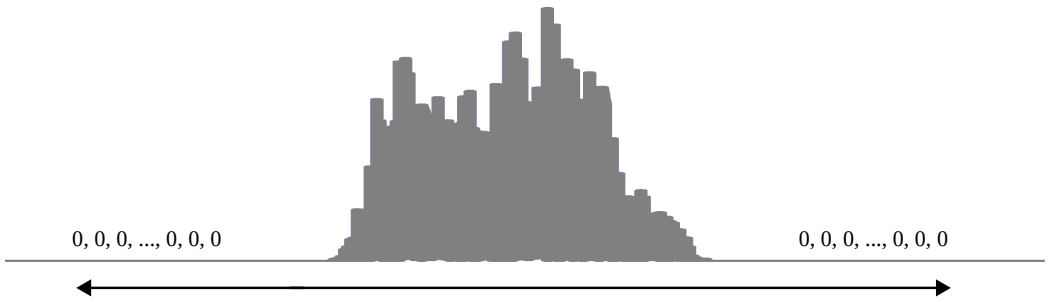

Figure 2.5. Zero padding before the final Fourier transform. The final length of the A-scan should be 16,384 complex values or 4096 × a scale factor.

The primary operation for reconstructing an A-scan into data that can be modified to form a viewable A-scan image is the fast Fourier transform [12]. An FFT computes the discrete Fourier transform and uses the following equation:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k \frac{n}{N}} \qquad k = 0, 1, 2, ..., N-1 \tag{2.6}$$

where x_n is a value in the time domain and X_k is the transformed value in the frequency domain [14].

2.3.7 Cropping

After the final Fourier transform step, the retinal information will only occupy a small portion of the final A-scan length, approximately two times the width of the original A-scan spectrum. Therefore, cropping is desired in order to save computational and memory resources.

Cropping is done for each A-scan. The top of the A-scan is cropped mainly due to the fact that the pixels in the area will typically have large optical artifacts. These artifacts result in high pixel values because it is close to the "coherence gate" of the OCT system, which is used to separate depth layers [15]. High pixel values will skew normalization step and create a dark image. The bottom of the A-scan is cropped as well in order to remove large areas with very little useful information due to sensitivity roll-off of the system and artifacts from residual dispersion. Cropping can be defined as:

$$A \setminus C := \{x \in A : x \notin C\} \tag{2.7}$$

where A is the set of pixels in the reconstructed A-scan and C is the set of pixels within the top and bottom sections to be cropped out. The resulting set includes all pixels that are elements of the reconstructed A-scan except those cropped out.

2.3.8 Converting the Complex Image to Intensity

Since an intensity image is desired, it is necessary to convert between the output of the Fourier transform into intensity values. The Fourier transform outputs complex values made up of a real component and an imaginary component in addition to the complex conjugate [16]. Intensity is calculated by multiplying a complex number by the complex conjugate. If a is the real part of a complex number and b is the imaginary part, then intensity, I, for each pixel can be calculated using the following equation:

$$I = (a + bi) \times (a - bi) = a^2 + b^2 \tag{2.8}$$

This is essentially squaring and adding the real and imaginary components, and therefore, having the complex conjugate stored in memory is unnecessary.

Another desired image uses decibel (dB) values. Once the intensity image is computed, decibel values can be computed with ten times the logarithm of the intensity value [17], as seen with the following equation:

$$I_{dB} = 10 \log_{10}(I) \tag{2.9}$$

2.3.9 Normalization

The values obtained from the intensity calculation can vary wildly in the positive real number domain. In order to display this information on the screen as an 8-bit grayscale image, it is necessary to normalize the data such that it fits in a range of integers between 0 and 255 inclusive. The formula for normalization is as follows:

$$I_N = 255 \left(\frac{I - min}{max - min} \right) \tag{2.10}$$

This is done once for every B-scan where min represents the minimum intensity value of an entire B-scan and max represents the maximum intensity value of that same B-scan. Normalization must also be done for the dB image with the dB minimum and maximum.

2.3.10 Retinal Tracking

An optional step when processing AO-OCT data is to do z-axis registration because of depth motion during the scan. There are a number of methods of z-axis registration, but only two methods were considered. The first method creates and aligns a weighted mean contour of the fast B-scan and the other method uses peak detection to find and align the retinal pigment epithelium (RPE) layer. A common

technique for the alignment is to shift each B-scan to a mean of the path lengths between several B-scans as shown in the following formula:

$$\mu = \frac{\sum_{i=0}^{n-1} w_i L_i}{\sum_{i=0}^{n-1} w_i} \tag{2.11}$$

where μ represents the mean shift needed, w represents the weighted mean, and L is the tracked point. The sums are from $i = 0$ to $n - 1$ scans [18].

3 PARALLEL COMPUTING

3.1 A Brief History of Supercomputing

Supercomputing began its start in the 1960s with Control Data Corporation and Seymour Cray. In 1964 the CDC 6600 could perform about 1 million floating point operations per second (MFLOPS), and by 1969 the CDC 7600 was ten times faster by increasing parallelism through the use of pipelining. The trend of making bigger and faster machines took off. In 1985, the Cray-2 could perform 1.9 billion floating point operations per second (GFLOPS) and later in the 1990s, the United States government developed ASCI Red. In 1996 ASCI Red was the first supercomputer to be able to performs over 1.6 teraFLOPS (TFLOPS) [19, 20]. The supercomputers of today can perform at the petaFLOPS level and above.

These machines, in their respective times, cost millions of dollars, took up a large space, and had enormous power requirements. One of the solutions to the cost and power requirements is the use of commodity hardware wired together into clusters. Machines in a cluster can compute tasks in parallel at relatively low cost. One of the issues with clusters is that communication between nodes becomes a bottleneck. If a task requires internode communication then the throughput on a cluster will be much less than a task that requires no communication [21]. A supercomputer or even a cluster will not fit into a small clinical environment and using such resources over a typical network will eliminate hopes of real-time computation due to latency [22].

3.2 General-Purpose Computing on Graphics Processing Units

Computing on a graphics processing unit (GPU) began in the early 2000s with programmable shaders. Developers began attempting to abstract these out to computing tasks other than calculating graphical effects.

At the heart of modern GPUs lies massively parallel interconnected hardware, which as a whole can compute trillions of operations per second without the power and size requirements of the supercomputers of the past.

3.2.1 NVIDIA and CUDA

In 2007, one major GPU manufacturer, NVIDIA (Santa Clara, CA), released a programming interface for GPUs called CUDA, which stands for Compute Unified Device Architecture. CUDA is an extension to the C, C++, and Fortran programming languages and is therefore easier for programmers than having to learn shader languages which can be complex [21, 23].

A program in CUDA works by calling functions on the GPU known as kernels, which execute across a number of parallel threads. These threads are grouped into blocks, and a group of blocks is known as a grid. Within each thread block and grid, each thread has access to built-in variables that contain the thread and block identification numbers. Threads have their own private registers and can access shared memory among threads in the same block. Accessing shared memory is slower than accessing registers. Threads can also access GPU global memory but there is a greater latency penalty for doing so.

3.2.2 Kepler Architecture

NVIDIA introduced the Kepler microarchitecture in 2012. Each card contains several streaming multiproccessors or SMs. NVIDIA calls these SMXs with Kepler as opposed to SMs to illustrate an improvement of functionality over the previous

microarchitecture, Fermi. This document focuses on Kepler and specifically the NVIDIA Tesla K20c and NVIDIA Titan Z graphics cards tested that contain variants of the GK110 GPU. On the GK110 chip, each SMX contains 192 CUDA cores, which can be used for single precision floating point and integer arithmetic operations [23].

When a CUDA kernel is launched, it is given parameters for the number of threads per block and number of blocks to launch. An SMX will then execute threads in groups of 32, called warps. In Kepler, threads within a warp can communicate with the shuffle command without suffering a latency penalty for using shared memory or the need to explicitly synchronize threads. The shuffle command tells a thread to read the values in the registers belonging to another thread [23].

The Tesla K20c contains 13 SMXs and the Titan Z contains two groups of 15 SMXs yielding a core count of 2496, 2880, and 2880, respectively. Combined in one full tower case, this achieves a theoretical maximum of 11.52 TFLOPS, which makes it faster and over ten thousand times cheaper than the top supercomputers at the turn of the century.

3.2.3 CUDA Intrinsic Functions

Mathematical functions can be performed using standard or intrinsic functions. Standard functions can be used on both the CPU and the GPU, while intrinsic functions can only be used on the GPU. Intrinsic mathematical functions map to fewer native instructions and thus will consume fewer clock cycles. The CUDA compiler may not always honor the *-use_fast_math* option to compile standard functions into their intrinsic counterparts. Explicitly using intrinsic functions in the source code guarantees that the compiler will generate the desired output. Note that certain intrinsic functions can be less accurate than their standard counterparts. A full list of both intrinsic and standard functions and their accuracy is available in the CUDA programming guide provided on the NVIDIA website [24].

4 PRIOR WORK

There are various approaches to performing the computations necessary to visualize OCT data. Methods vary in accuracy of the resulting images and also in execution time. These approaches will differ depending on the type of OCT. The presented sections will focus primarily on currently used methods that are built and used for AO-OCT.

4.1 CPU Based Processing

Data generated by AO-OCT can be processed using just a CPU, but computation speeds are limited when the machine has only a single, multi-core CPU. There are multiple ways to compute AO-OCT data on a CPU. One can use off-the-shelf mathematics software (e.g. MATLAB) or create a single or multithreaded application.

4.1.1 MATLAB®

One of the simplest approaches to performing the necessary computations on OCT data is to save the data to a binary file and process it later using MATLAB® by MathWorks (Natick, MA). MATLAB is an easy-to-code high-level language that allows one to quickly implement algorithms. Although MATLAB offers ease of use and fast implementation, it suffers from slow execution time [25].

4.1.2 C++

An alternative to MATLAB is to use C++. C++ is a compiled language and through using the standard math libraries and the popular FFT library, FFTW, it is possible to implement a solution using the processing method studied previously [3, 26].

4.2 GPGPU Based Processing

Many researchers are turning to GPGPU solutions for processing OCT data due to obvious speed and other advantages. Typically, they use either a single CPU thread to control a GPU or multiple CPU threads each with their own work flow [1, 2, 4, 5].

4.2.1 Single CPU Thread Approach

Our own prior attempt at a GPGPU-based solution using a single CPU thread to control the GPU lacked the use of a Hann function to taper the ends of an A-scan down to zero. Unwanted oscillations will appear in the resulting OCT image if the ends of each A-scan do not reach zero. The prior attempt lacked the zero padding before the final FFT step and required the use of two GPUs for the processing of a single B-scan. Zero-padding results in increased pixel resolution in the image domain. The process was designed for a single camera input and lacked the ability to scale to multiple cameras [2].

Other single CPU threaded methods have been used to control a single GPU and many times these methods will skip or otherwise change processing steps, resulting in lower accuracy in order to gain speed [1, 4].

4.2.2 Multithreaded CPU Approach

In the past, one multithreaded implementation for computing OCT data achieved very fast computation times of over 1.1 million A-scans per second [5]. It was designed for a single camera and single GPU. However, their implementation used an FFT size of 2048, and therefore, it may not be as accurate as larger FFT sizes such as the aforementioned 16,384 pixel size. It works by grouping B-scans into large batches and then launches a new CPU thread to control the GPU workload for each batch. Without batching together multiple B-scans, the throughput is roughly 0.9 million A-scans per second. It should be noted that with this approach DC bias is not recalculated per B-scan and the final results are not transferred back to system memory as it would reduce throughput. This prevents later analysis of the data [5].

A multithreaded solution that utilized two GPUs used one GPU for B-scan processing and a second for display. This method was able to achieve 186,000 A-scans per second with an FFT size of 2048 [6].

There exists no multithreaded or single-threaded multiple GPU approach that we know of that automatically scales B-scan processing to each GPU and handles up to four cameras.

5 IMPLEMENTATION AND APPROACH

5.1 Problem

The AO-OCT system in Bloomington has four CMOS sensors, each of which captures an A-scan sequentially. This leads to an interlaced fast B-scan. The data in each A-scan must be adjusted based upon which sensor it came from. In addition, the accuracy desired causes an increase in complexity and thereby increasing computation time.

5.2 Design

The need to process AO-OCT data will remain long after the creation of this document. In addition, technology is ever-evolving and with that comes new methods or algorithms to compute and generate data. With this in mind, the very design of the application becomes important. In order to allow the software to easily adapt as it evolves to meet future needs, design patterns were used to allow a level of software extensibility [27].

5.2.1 Strategy Pattern

In order to best interface with control and display software being independently developed by the team in IU Bloomington, the strategy pattern was chosen to make different algorithms interchangeable [27].

The structure of the strategy pattern is shown in Figure 5.1. It consists of a context that will keep a reference to the strategy object, a strategy that declares the interface to the defined algorithms, and the concrete strategies that implement the algorithms [27].

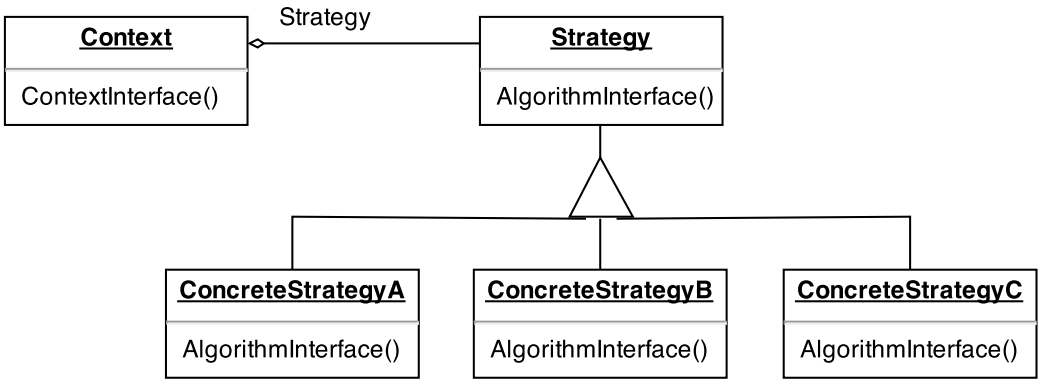

Figure 5.1. The strategy pattern that allows multiple algorithms to be used interchangeably through the same interface [27].

For AO-OCT data processing on the GPU, the context of the strategy is the software written at IU Bloomington. The strategy itself is an abstract class called CuAOOCT, which is short for "CUDA AO-OCT." The strategy contains an interface called *run()*, which exists in all of the possible concrete strategies. With this design, new algorithms for processing can be added at any time. Switching between the algorithms involves a change of only a single line of code in the IU Bloomington acquisition software. A diagram for this structure appears in Figure 5.2, and the structure allows for algorithm customization by virtue of the design pattern itself.

5.2.2 Proxy Pattern

One of the vital third party libraries is NVIDIA's CUDA FFT (CuFFT) library, which plans and executes FFTs on the GPU. The major problem with this library is that it does not allow the use of FFTs on multiple GPUs by an application unless the GPUs exist on the same graphics card [14]. This causes a major issue when

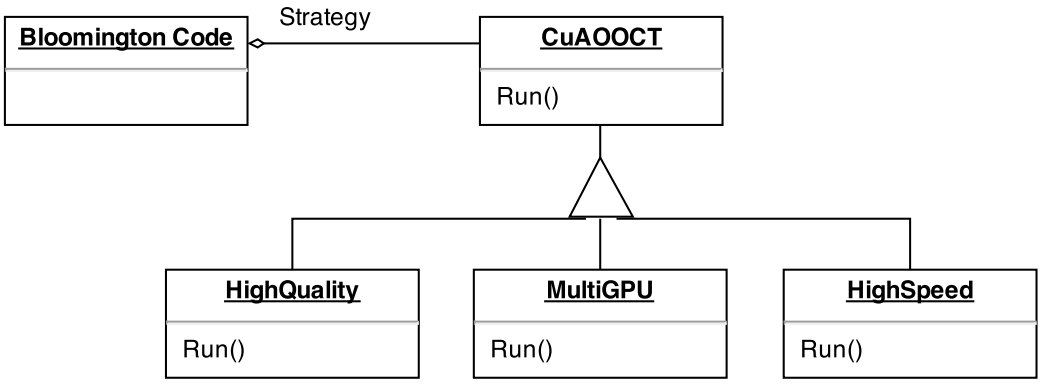

Figure 5.2. The strategy pattern implemented for processing on the GPU. A concrete strategy for computing on the CPU can utilize the same interface created by the CuAOOCT strategy [27].

attempting to scale the processing to execute on multiple GPUs. In order to alleviate this issue, each GPU must be controlled by a separate CPU process.

Instead of rewriting either the acquisition code from IU Bloomington or modifying the AO-OCT processing code created to run on a single GPU, a single class can act as a proxy between the two pieces of code and manage GPU resources available on the machine. This is where the proxy pattern proves beneficial.

The proxy pattern, shown in Figure 5.3, allows for a way to control access to an object by acting as a placeholder [27]. Using the proxy pattern, a proxy was created called multiGPU, which is a subclass of CuAOOCT. This allows the use of the same common interface created when implementing the strategy pattern.

The multiGPU concrete strategy, acting as a proxy, shown in Figure 5.4, will control one separate CPU process with an instance of the HighQuality concrete strategy subclass for each GPU and will distribute work accordingly to each of the objects. The concrete strategy chosen for the proxy is a matter of user preference. This thesis

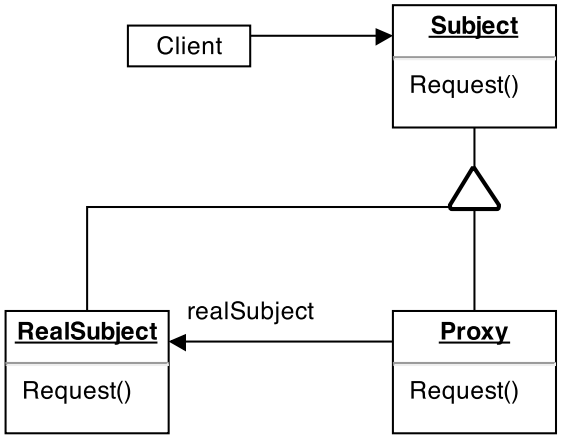

Figure 5.3. The proxy pattern: This pattern uses a placeholder called a proxy to control access to another object [27].

will focus on a concrete strategy subclass called HighQuality that provides all of the processing steps through normalization.

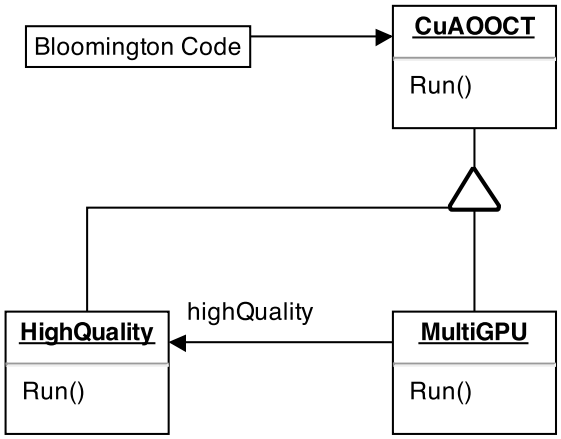

Figure 5.4. The structure of an implementation of the proxy pattern using the multiGPU class as a proxy for the HighQuality class creating a private object named "highQuality" [27].

5.3 Algorithms

A-scans from the AO-OCT system are aggregated to form an entire fast B-scan before being sent to the GPU for processing. A-scans are placed sequentially into an array residing in page-locked memory (also called pinned memory). When a B-scan is complete, pointers to the page-locked input array and an unsigned character output array are packaged in a B-scan object. Then, a pointer to the B-scan object is sent over to the CuAOOCT object for processing. Processing on the GPU is accomplished through a series of kernels that transform the raw data into a viewable image. These kernels are meant to be reused and customized for various methods of computing AO-OCT data. Once processed, a set of B-scans can be used to create an *en-face* view of the retina.

One of the limitations in the current implementation of the DC subtraction, k-space mapping, and phase correction algorithms is that they were created to work with one, two, or four cameras. To work with any other number of cameras, those specific algorithms must be rewritten in a new concrete strategy.

5.3.1 Cast to Float

Spectra data from the CMOS sensors come in as 16-bit (short) unsigned integers. Many of the computations, built-in libraries, and architecture of the GPU are built to work with floating point values, and therefore, the data must be cast to floating point. This is not as straightforward as it seems because in order to increase efficiency of the movement of data from/to global GPU memory to be cast into a float, one must cast an unsigned short integer pointer to a ushort4 data type pointer that will be used to grab four unsigned short values at a time and then copy the values to the GPU registers. From there, the values can be cast into floating point values and stored into a float4 data type. This can then be transferred back to GPU global memory, but will require twice the space the original short integers have taken up in system memory. If the dataset is not a multiple of four then the remaining values can

be picked up by an if statement. The general algorithm for the kernel using ushort4 vector loads and float4 vector stores is shown in Algorithm 5.1:

Algorithm 5.1 Cast To Float (inputArray, outputArray, size)

```
1: idx ← blockIdx.x × blockDim.x + threadIdx.x
2: iterations ← size ÷ 4
3: for i; while i < iterations in steps of (blockDim.x × gridDim.x) do
4:     ushort4 input ← (ushort4*) inputArray[i]          ▷ Move data to registers
5:     float4 ouptut ← input cast to floating point
6:     (float4*) outputArray[i] ← output                 ▷ From registers to global memory
7: end for
8: i ← (idx ÷ 4) × 4                                     ▷ Utilize truncation from integer division
9: if i < size then
10:    outputArray[i] ← (float)inputArray[i]
11: end if
```

5.3.2 Hann Filter

The Hann filter is only executed for A-scans of a length less than 832 by design. Furthermore, if it executes, it will only modify values within user-defined areas at the front and back of the spectrum called "cuts". The kernel for the Hann filter is straightforward and displayed in Algorithm 5.2. If the data is within the domain of the cuts then apply the filter, otherwise skip.

Algorithm 5.2 Hann Filter (input, width, size, cut)

```
1: endCut ← width − cut
2: i ← blockIdx.x × blockDim.x + threadIdx.x
3: for i; while i < size in steps of (blockDim.x × gridDim.x) do
4:     row ← i ÷ width                                          ▷ Determine row
5:     col ← i % width                                          ▷ Determine column
6:     index ← row × width + col                                ▷ Calculate index
7:     if col < cut then
8:         temp ← input[index]                                  ▷ Move to register
9:         high ← input[row × width + cut]                      ▷ Move to register
10:        input[index] ← 1/2 × ( 1 − cos(( 2π × temp ) ÷ width )) ÷ high
11:    else if col > endCut then
12:        temp ← input[index]                                  ▷ Move to register
13:        high ← input[((row + 1) × width) − cut]              ▷ Move to register
14:        input[index] ← 1/2 × ( 1 − cos(( 2π × temp ) ÷ width )) ÷ high
15:    end if
16: end for
```

In the algorithm, duplicate instructions exist within the if statement and else if. This is intentional in order to only execute those items within warps which have threads that meet the conditional argument; otherwise the entire warp will return from the function without performing unnecessary work.

5.3.3 Transpose and Reduce

In order to calculate the average across all A-scans per CMOS sensor, the data must first be transposed so that GPU global memory accesses are coalesced. An efficient matrix transpose algorithm was developed by NVIDIA and is available at their website [28]. One major difference in our implementation is that the data set is zero-padded on the tail end such that when transposed will produce rows which have a length that is a power of two. Once transposed, the data is still interlaced, but in columns instead of rows. Each column of the B-scan now represents output from a specific sensor. An array of averages must then be calculated. The array must be the size of the number of CMOS sensors multiplied by the original A-scan length. The

value of each entry in the array is equal to the sums of all values in that row, for that sensor, divided by the number of A-scans per B-scan belonging to that sensor.

To compute this on the GPU efficiently, threads need to communicate within each warp. Since up to four sensors are supported, it must also be able to return values in groups of four. At the warp level this is implemented by a reduction that shuffles down the values similar to folding a piece of paper in half. This process stops when only one value remains. This process is illustrated in Figure 5.5. Since GPU global memory transfers are expensive in terms of the run time, it is more efficient to transfer four values at a time. Therefore this was done using a float4 data type containing four floating point values.

Once the reduction completes, the resulting float4 is further reduced to match the number of sensors. The unused values are set to zero. Each warp will handle 128 floating point values. Algorithm 5.3 depicts this warp reduction. The number of A-scans per B-scan will vary, and thus, to guarantee this algorithm runs as expected with four values at a time, the data was zero-padded before being transposed. This creates a row length that is always going to be evenly divisible by four.

At the block level when each warp returns a float4, the first thread of each warp will write the float4 to shared memory. The threads in the block are then synchronized, and thread zero will reduce the values from shared memory instead of calling the warp reduce function as the number of operations is very small. After being reduced, the number of rows from each sensor is calculated and the components of the float4 are divided by the appropriate numbers. The row (A-scan) calculations rely on truncation as a result of integer division. The original height is not guaranteed to divide equally among the sensors so the number of A-scans for each sensor must be calculated individually. The height is added to the number of sensors and the sensor number plus one is subtracted from this value. The integer result of a division of this number and the number of sensors will equal the number of A-scans from that sensor per B-scan. The average is calculated by dividing the reduced sums by the number of

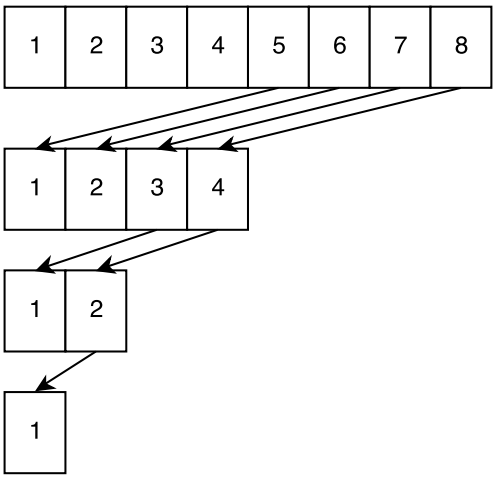

Figure 5.5. The basic warp reduce. Starting with 32 threads (not shown) a reduction is performed by folding the values over until a single value is left. The value in this case is a float4 containing four floating point values.

A-scans for that sensor. This is better illustrated in Algorithm 5.4. The averages are stored in an array that has float4 values and is equal to the A-scan original length.

As an example, if an A-scan length is 832 pixels (columns) and there are 240 A-scans per B-scan (rows) then the B-scan will be zero-padded to contain 256 rows. This data is then transposed, yielding 256 columns and 832 rows. Each row will be computed by a single block in CUDA. Since each thread will handle four values, only two warps are needed for each block to reduce the 256 values in each row. Once reduced, the values of the float4 are each divided by 60, or 240 A-scans $\div$ 4 cameras, because each of the four cameras acquired the same number of A-scans in this example. This calculated average is then placed into an array of float4 values with a length of 832 for use with DC subtraction.

Algorithm 5.3 Warp reduce by four where stride is the number of sensors

```
1:  function warpReduceBy4(value, stride)
2:      laneID ← threadIdx.x % 32                    ▷ Determine the current lane
3:      for i ← 16; while i >0 i ← i ÷ 2 do
4:          float4 n ← the shuffled down float4 value from i threads higher than laneID
5:          if laneID ≤ i then
6:              value ← value + n                    ▷ for each of the four float components
7:          end if
8:      end for
9:      if  stride < 4 then
10:         value.x ← value.x + value.z
11:         value.y ← value.y + value.w
12:         value.z ← value.w ← 0
13:         if stride < 2 then
14:             value.x ← value.x + value.y
15:             value.y ← 0
16:         end if
17:     end if
18: return value
19: end function
```

Algorithm 5.4 Block Reduce

```
1: function BLOCKREDUCE(input, output, stride, width, height)
2:     warpid ← threadIdx.x ÷ 32                          ▷ Determine which warp
3:     warpsPerBlock ← blockDim.x ÷ 32                    ▷ Determine warps per block
4:     N ← width × blockDim.x                             ▷ Determine size
5:     Declare float4 final[16]
6:     idx ← threadIdx.x + blockDim.x × blockIdx.x
7:     for idx; while idx < N; idx ← idx + blockDim.x × gridDim.x do
8:         declare float4 sums and set all components to zero
9:         value ← input[idx]
10:        first ← warpReduceBy4(value, stride)
11:        if threadIdx.x % 32 == 0 then
12:            final[warpid] ← first
13:        end if
14:        Synchronize Threads
15:        if threadIdx.x == 0 then
16:            for i ← 0; while i < warpsPerBlock; + + i do
17:                sums ← sums + final[i]                 ▷ For each component
18:            end for
19:            numrows ← height + stride
20:            sums.x ← sums.x ÷ ((numrows − 1) ÷ stride)
21:            sums.y ← sums.y ÷ ((numrows − 2) ÷ stride)
22:            sums.z ← sums.z ÷ ((numrows − 3) ÷ stride)
23:            sums.w ← sums.w ÷ ((numrows − 4) ÷ stride)
24:            output[idx ÷ blockDim.x] ← sums
25:        end if
26:    end for
27: end function
```

5.3.4 Subtract DC

Using the array of averages obtained from the previous step, the DC bias can be subtracted out from the rest of the signal. Each thread must determine which row, column, and sensor it is working with in order to subtract the appropriate values from the array of averages as shown in Algorithm 5.5.

Algorithm 5.5 Subtract DC Bias

```
1: function SUBTRACTDC(input, output, averages, stride, size, offset)
2:     i ← blockIdx.x × blockDim.x + threadIdx.x
3:     for i; while i < size in steps of (blockDim.x × gridDim.x) do
4:         row ← i ÷ width                                  ▷ Determine row
5:         col ← i % width                                  ▷ Determine column
6:         camera ← row % stride                            ▷ Determine which sensor
7:         out_index ← row × width + col                    ▷ Calculate index
8:         output[out_index] ← input[i] − averages[(col × 4) + camera]
9:     end for
10: end function
```

5.3.5 k-Space Interpolation and Dispersion Compensation

Similar to the DC subtraction step, it is important for each thread to know to which row, column, and sensor any given pixel belongs because the interpolation and dispersion compensation values are calculated separately for each sensor.

The first step shown in Algorithm 5.6 was to create a k-space index table for each of the sensors. Since this only needs to be done one time per sensor, the function call was placed in the CuAOOCT class constructor. In order to speed up linear interpolation to k-space, this function finds the index just to the left of the desired interval and stores it in an array for later use during the B-scan processing.

Algorithm 5.6 Build k-space index table

```
1: function BUILD_kSPACE_INDEX(index, new_wavelengths, interp_buffer, length)
2:     minK ← 2π ÷ new_wavelengths[0]
3:     maxK ← 2π ÷ new_wavelengths[length − 1]
4:     interval ← (maxK − minK) ÷ (length − 1)
5:                                   ▷ Create an evenly space interpolated buffer
6:     for i ← 0; while i < length in steps of 1 do
7:         interp_buffer[i] ← 2π ÷ (minK + i × interval)
8:     end for
9:                                   ▷ Create index table for faster linear interpolation
10:    idx ← 0
11:    for  i ← 0; while i < length in steps of 1 do
12:                    ▷ Find an appropriate index to the left of the desired interval
13:        while new_wavelengths[idx+1] > interp_buffer[i] do
14:            idx ← idx + 1
15:        end while
16:        index[i] ← idx
17:    end for
18: end function
```

The next step, which happens during the B-scan processing, is the actual k-space mapping and dispersion compensation. These were combined into one step in order to save an intermediate step of reading and writing to global GPU memory.

The implementation pseudocode presented in Algorithm 5.7 is simply a means of computing a linear interpolation similar to the built-in MATLAB function *interp1()*. The computations on this kernel are not complex so this is a bandwidth bound kernel. Dispersion compensation is done by simply multiplying the real result of the k-space mapping with the appropriate real and imaginary entries in the phase correction data set.

Algorithm 5.7 k-space and Dispersion Compensation

```
1: function KSPACE_AND_DISPERSION(x, y, interp_buffer, index, output, num_lines,
   num_cameras, phase, maxline_scale)
2:     size ← num_lines × 4096
3:     i ← blockIdx.x × blockDim.x + threadIdx.x
4:     for i; while i < size in steps of (blockDim.x × gridDim.x) do
5:         row ← i ÷ width                                          ▷ Determine row
6:         col ← i % width                                          ▷ Determine column
7:         camera ← row % num_cameras                               ▷ Determine which sensor
8:         if i is within the first two or last two columns then
9:             y[i] ← 0
10:        end if
11:        idx ← index[col]                                         ▷ Move data to registers
12:        offset ← row × 4096                                      ▷ Determine input offset
13:        cam_offset ← camera × 4096          ▷ Determine offset for correction data
14:        sum ← offset + idx        ▷ Place sum in register to speed later operations
15:        y2 ← y[sum+1]                                            ▷ Move data to registers
16:        y1 ← y[sum]                                              ▷ Move data to registers
17:        x2 ← x[cam_offset + (idx + 1)]                           ▷ Move data to registers
18:        x1 ← x[cam_offset + idx]                                 ▷ Move data to registers
19:        m ← (y2−y1)/(x2−x1)                                      ▷ Determine slope
20:        kspace ← m × interp_buffer[cam_offset+col] + y1 − m × x1        ▷ Solve
21:        out.x ← phase[cam_offset+col].x × kspace                        ▷ real
22:        out.y ← phase[cam_offset+col].y × kspace                        ▷ imag
23:                    ▷ Prepare offset for 4096 × maxline scale factor for FFT size
24:        offset ← (row × 4096 × maxline_scale) + (2048 × (maxline_scale − 1))
25:        output[offset+col] ← out                  ▷ output to zeroed, scaled, array
26:    end for
27: end function
```

5.3.6 Intensity

Converting between complex values and intensity is straightforward. From the complex values, the real and imaginary values are brought into the registers; they are squared, added, and then transferred back to global memory. For every floating point operation, a CUDA floating point intrinsic function is used in order to minimize the clock cycles consumed. Cropping is also handled by the intensity kernel in the actual implementation, but has been omitted for clarity in Algorithm 5.8.

Algorithm 5.8 Intensity Kernel

```
1: function INTENSITY(input, output, log_output, width)
2:     i ← blockIdx.x × blockDim.x + threadIdx.x
3:     for i; while i < size in steps of (blockDim.x × gridDim.x) do
4:         row ← i ÷ width
5:         real ← input[ 2 × i ]                          ▷ Move to registers
6:         imag ← input[ 2 × i + 1]                       ▷ Move to registers
7:         out_index ← row × width
8:         real ← real^2
9:         imag ← imag^2
10:        temp ← real + imag
11:        output[out_index] ← temp          ▷ Write outputs to global memory
12:        log_output[out_index] ← 10 × log(temp)
13:    end for
14: end function
```

5.3.7 Find Minimum and Maximum

In order to properly normalize an image, a minimum and maximum float2 value needs to be either obtained from the user, or determined based upon the current data set. When values are not given by the user, a kernel executes to find the minimum and maximum values.

The kernel created for this purpose utilizes inter-thread communication within each warp. The warps will determine the minimum and maximum from that warp and write it out to shared memory. That shared memory array is reduced again in a single warp and this warp will provide the final output per block. The block outputs are atomically compared to the final output for the function. When complete, the final output for the function will contain the minimum and maximum values for the entire data set.

It is more efficient to read and write from/to global memory only once to find both the minimum and maximum than to do the identical work for both tasks. To accomplish this, a number of built-in CUDA functions had to be rewritten to suit the needs of the operation. For inter-thread communication, a shuffle down function was written to carry a float2 data type between threads in order to compare minimum and

maximum values. This utilizes the built-in floating point minimum and maximum intrinsic operations as well as the built-in shuffle down (*__shfl_down()*) function for 32-bit values as shown in Algorithm 5.9.

Algorithm 5.9 float2 shuffle down to move a minimum and maximum value

```
1: function __SHFL_DOWN(var, srclane, width)
2:     var.x ← minimum of var.x or shuffled value
3:     var.y ← maximum of var.y or shuffled value
4: return var
5: end function
```

At the warp level, the reduction is a simple for-loop as in Algorithm 5.10, which makes a call to the custom *__shfl_down()* that was defined in the previously shown Algorithm 5.9. Since each warp is 32 lanes wide, a shuffle down from the last 16 lanes to the first 16 lanes yields 16 minimum and maximum values. This is shuffled down further from the last 8 to the first 8 and so on until there is a single minimum and maximum float2 value for the entire warp. This is the same concept depicted in Figure 5.5. The custom *__shfl_down()* algorithm makes two calls to the built-in *__shfl_down()* and therefore requires fewer clock cycles for each of the five iterations of the loop and no reading and writing to shared memory. The loop itself is unrolled in the actual implementation so that clock cycles are not needlessly wasted.

Algorithm 5.10 Warp reduction to find minimum and maximum

```
1: function WARPREDUCEMINMAX(value)
2:     for offset ← 16; while offset > 0; offset← offset ÷ 2 do
3:         value ← __shfl_down(value, offset)
4:     end for
5: return value
6: end function
```

At a high level, the block algorithm calls the warp reduce function for all threads, synchronizes between the threads, and writes the results from each warp to an array in shared memory. This array is then used with another warp reduce call to find the minimum and maximum between all the warps within the block. When this float2

value is returned, it is atomically compared with values returned from other blocks so that a minimum and a maximum from the entire data set can be determined. The high level overview is displayed in Algorithm 5.11.

Algorithm 5.11 Block reduction to find minimum and maximum

```
1:  function BLOCKREDUCEMINMAX(value)
2:      lane ← threadIdx.x % warpSize                    ▷ Determine which lane
3:      warpid ← threadIdx.x ÷ warpSize                  ▷ Determine which warp
4:      value ← warpReduceMinMax(value)
5:      if lane is zero then                  ▷ Lane zero holds the resulting value
6:          SharedArray[warpid] ← value
7:      end if
8:      Synchronize Threads
9:      if threadIdx.x < blockDim.x ÷ warpSize then          ▷ Warp existed
10:         value ← SharedArray[lane]
11:     else                                                 ▷ Warp did not exist
12:         value ← min/max default values
13:     end if
14:     if warpid is zero then
15:         value ← warpReduceMinMax(value)
16:     end if
17: return value
18: end function
```

The actual implementation differs from what is shown in Algorithm 5.11 in that some mathematical operations are done in a more efficient manner for the GPU in order to speed up computation. For example, because the warp size is known to be a power of two, it is faster to determine the lane using a binary AND operation such as $threadIdx.x$ & $(warpSize - 1)$. Likewise, division is also accomplished in a more efficient way by using a right shift.

5.3.8 Normalize

The normalization kernel normalizes values between 0 to 255 so they can appear as 8-bit grayscale image. By using the floating point intrinsic *__saturatef()* command in CUDA, values can be normalized between 0 to 1 and then scaled to 255. The

minimum is negated as it is brought in from global memory in order to use a floating point intrinsic multiplication to compute the result in one cycle and to keep that step outside of the for loop.

Algorithm 5.12 Normalization

```
1:  function NORM(input, size, min, max)
2:      minimum ← -1 × min                                  ▷ Move value to register
3:      maximum ← max                                       ▷ Move value to register
4:      i ← blockIdx.x × blockDim.x + threadIdx.x
5:      for i; while i < size in steps of blockDim.x × gridDim.x do
6:          inpt ← input[i]                                 ▷ Move value to register
7:          numerator ← inpt + minimum
8:          denominator ← maximum + minimum
9:          normalized ← saturate(numerator ÷ denominator)  ▷ Clamp
10:         output[i] ← normalized × 255                    ▷ Scale
11:     end for
12: end function
```

5.4 Job Scheduling on Multiple GPUs

In an effort to increase throughput, multiple GPUs can be used to process AO-OCT data. As noted previously, a proxy pattern is used to control and schedule multiple objects instantiated from the HighQuality class. Each object controls a separate GPU. One of the issues that arises from this set-up is that not all GPUs have the same number of cores. It is important to know the number of cores on each GPU since the number of cores often determines throughput. A common set-up is to have one graphics card for display purposes and a GPU accelerator for computations, such as an NVIDIA Tesla GPU.

If a machine has multiple GPUs, say a high-throughput and a low-throughput GPU, then it stands to reason that if work is split evenly between the two GPUs, in a round robin fashion, and executed in parallel, the time for execution will only be as good as the work done on the low-throughput GPU. Therefore, in order to mitigate this fact, we must split the work unevenly between the two GPUs, giving more work to the high-throughput GPU. This splitting of work must be done dynamically to

maintain flexibility, improve throughput, and allow future GPUs to be used. One way to do this is a weighted round robin approach.

The first step in a weighted round robin approach to this problem is to determine how many cores each GPU has and assign a weight to each GPU based on this number as a ratio of the total number of cores in the system. A queue is then built based on this weight system in a round robin fashion. Psuedocode depicting this is shown in Algorithm 5.13. The queue stores GPU ordinal numbers in an amount equal to the ratio of cores as a percentage. The MultiGPU object assigns each B-scan to the child thread controlling the GPU with the ordinal number at the front of the queue. As ordinal numbers are popped from the queue, they are pushed to the back of the queue to recycle them in the same sequence generated by Algorithm 5.13.

Algorithm 5.13 Weighted Round Robin

```
1: declare vector cores
2: declare queue
3: total ← 0
4: queue_size ← 0
5: gpu_num ← 0
6: for each gpu i do                                          ▷ Get core count
7:     cores.push ( number of cores for gpu i )
8:     total ← total + number of cores for gpu i
9: end for
10: for each gpu i do                                         ▷ Determine weights
11:     cores[i] ← ( cores[i] ÷ total ) × 100
12:     queue_size = queue_size + cores[i]                    ▷ Keep track of future queue size
13: end for
14: for i ← 0; while i < queue_size in steps of 1  do         ▷ Create initial queue
15:     while (cores[gpu_num] ≤ 0) do                         ▷ Move to valid index if empty
16:         gpu_num ← ( gpu_num + 1 ) % cores.size()
17:     end while
18:     if cores[gpu_num] > 0 then                            ▷ Fill queue
19:         queue.push(gpu_num)
20:         cores[gpu_num] ← cores[gpu_num] − 1
21:     end if
22:     gpu_num ← (gpu_num + 1) % cores.size()                ▷ Increment gpu
23: end for
```

It should be noted that the load on a GPU cannot be measured on consumer graphics cards with the current NVIDIA drivers. The weighted round robin load balancing method is dynamically created at run-time based on hardware specifications returned from a call to the NVIDIA driver.

5.5 Workflow

The typical workflow for processing on the GPU involves the IU Bloomington code for acquisition, which is treated as a black box by design, packaging data into individual B-scans. A pointer to each B-scan is sent to the cuAOOCT object, and the data is processed. Since the design uses the strategy pattern, the method in which the B-scans are processed can easily change.

When using the cuAOOCT subclass MultiGPU as both a concrete strategy and a proxy for the cuAOOCT subclass HighQuality as shown in Figure 5.4, the user can scale the HighQuality workflow to multiple GPUs. The process will launch one child CPU process per GPU and assign work based on the number of cores present on the respective GPU. Each of the child processes has its own instance of a HighQuality object. The HighQuality object will pipeline and process multiple B-scans at a time within CUDA streams. The number of streams is configurable by the user up to 32. The parent and child CPU processes are synchronized through the use of events. Since there are no dependencies between GPUs, synchronization between the child CPU processes is not necessary. A diagram of such a workflow is shown in Figure 5.6. Data flow within the diagram begins with four cameras controlled by software at IU Bloomington, which places interleaved A-scans into pinned memory. Once an entire B-scan is captured, a pointer to that B-scan will be sent to the MultiGPU object. This object will assign the work to one of its child processes, each containing an instance of the HighQuality class. High level psuedocode of the *run()* function of the HighQuality class is shown in Algorithm 5.14

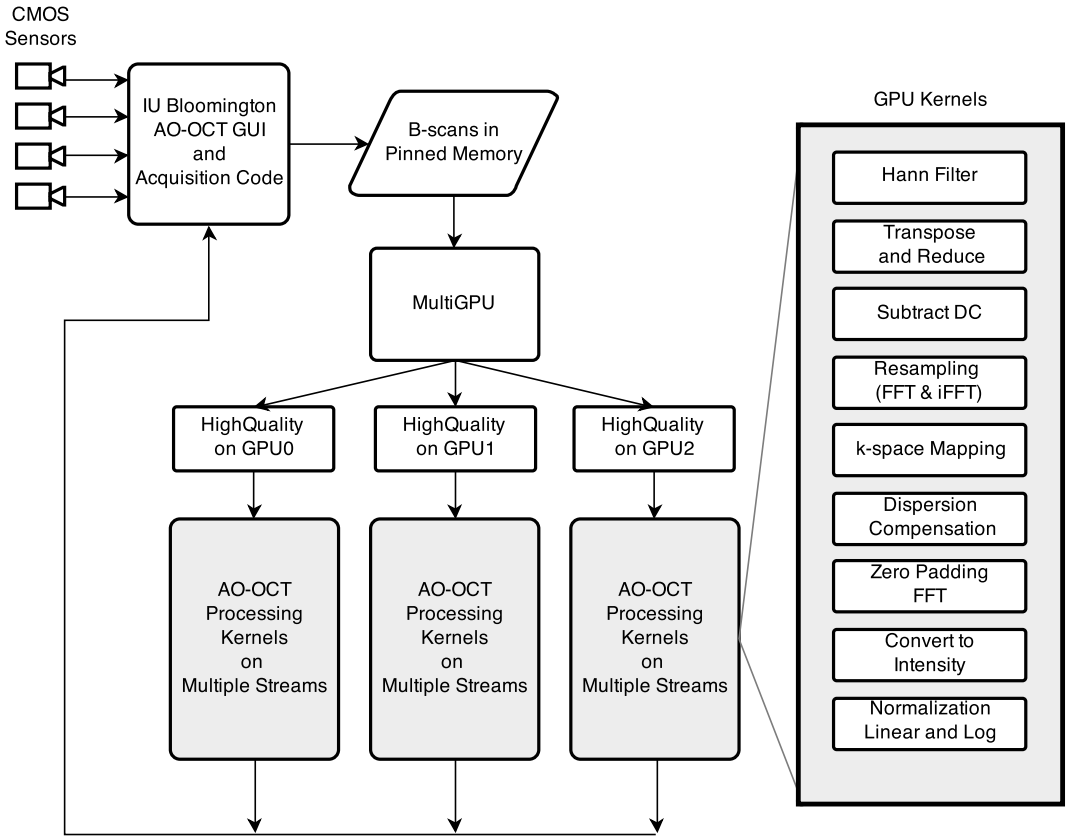

Figure 5.6. A flow chart diagramming AO-OCT processing using multiple CPU threads each of which controls a separate GPU. The processing pipelines on each GPU are launched in separate streams allowing concurrent B-scan processing on each GPU. The number of GPUs present is determined at run time.

Algorithm 5.14 Run function of the HighQuality class.

```
1: function RUN(Bscan * bscan)
2:     Get device pointer for the B-scan's short int original data
3:     CAST_TO_FLOAT(bscan->getOrigData(), orig_data_float, size)
4:
5:     if aLineLength < 832 pixels then
6:         HANN_FILTER(orig_data_float)
7:     end if
8:
9:     aLinesPerFrame_pow2 ← NEXTPOW2(aLinesPerFrame)
10:    TRANSPOSE_DATA(orig_data_float, transposed_data,aLineLength,
   aLinesPerFrame_pow2)
11:
12:    REDUCE(transposed_data, averages_for_DC_sub, numCameras, aLineLength,
   aLinesPerFrame)
13:
14:    SUBTRACT_DC(orig_data_float, subDC_out, averages_for_DC_sub,
   numCameras, aLineLength, size)
15:                                  ▷ Execute FFT and iFFT to resample to 4096
16:    CUFFTEXECR2C(cuForward, DCsub_out, FFT_out)
17:    CUFFTEXECC2R(cuInverse, FFT_out, iFFT_out)
18:
19:    KSPACE_AND_DISPERSION_COMP(newWavelengths, iFFT_out,
   d_interp_buffer, d_kspace_index, kspace_out, aLinesPerFrame, num_cameras,
   d_phase_correction)
20:                                                    ▷ Reconstructive FFT
21:    CUFFTEXECC2C(cuForwardComplex, kspace_out, final_FFT_out,
   CUFFT_FORWARD)
22:    INTENSITY(final_FFT_out, intensity_out, log_intensity_out, 4096*scale,
   4096*scale*aLinesPerFrame)
23:
24:    new_width ← (2 * aLineLength - crop_top) - crop_bottom
25:    MIN_MAX(intensity_out, minmax, new_width * aLinesPerFrame)
26:    MIN_MAX(log_intensity_out, log_minmax, new_width * aLinesPerFrame)
27:
28:    NORMALIZE(intensity_out, bscan->getDevNormIntensityImage(), new_width
   * aLinesPerFrame, minmax)
29:    NORMALIZE(log_intensity_out, bscan->getDevLogNormIntensityImage(),
   new_width * aLinesPerFrame, log_minmax)
30: end function
```

6 RESULTS

Comparisons of raw output generated by MATLAB, C++ running on the CPU, and GPU show identical results for one, two, and four sensors. A comparison was also done for each of the implemented GPU kernels for verification of the respective outputs. It is important to note that MATLAB normalizes FFT results, whereas FFTW and the cuFFT library outputs are scaled by N. This must be adjusted for before a direct comparison can be made. An example output of a processed B-scan appears in Figure 6.1.

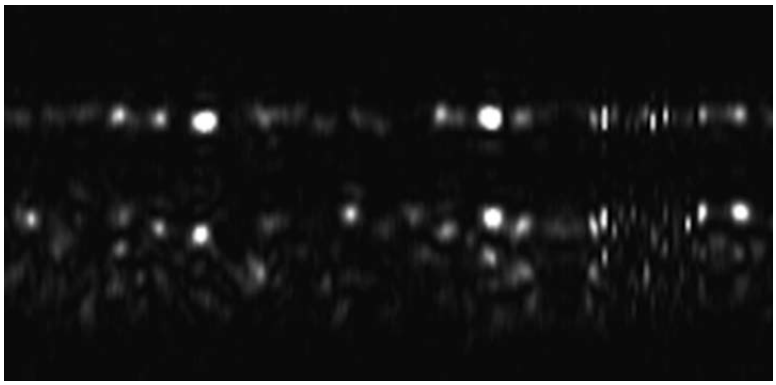

Figure 6.1. Example of a processed B-scan.

Scaling computation of multi-camera AO-OCT to multiple GPUs was successful and shows a near linear reduction in computation time as the number of GPU cores increases, as shown in Table 6.1. Run time for processing done on a CPU through MATLAB is given for a comparison. Utilizing three GPUs shows a speed up of over 653×.

Table 6.1
Run times on multiple GPUs on a sample data set 832×240×240×11. This table reveals scaling with an increased number of cores. The CPU processing run time in MATLAB is also given for comparison.

GPUs	**# of Cores**	**Time (sec)**
1 GK110B(Titan Z)	2880	2.869
2× GK110B(Titan Z)	5760	1.493
2× GK110B(Titan Z) + 1 GK110 (Tesla K20c)	8256	1.073
CPU		
MATLAB	-	721.000

The 4096 pixel scale factor for the final FFT step can dramatically affect run times as shown in Table 6.2. This shows that there is a computational penalty for increased accuracy.

Table 6.2
Run times with different FFT scale factors for a sample data set 832×240×240×11 and FFT sizes of 4096×Scale factor on an Nvidia Tesla K20c and both GPUs on an Nvidia Titan Z.

Scale Factor	**Time (sec)**	**A-scans per sec**
1	0.541	1.16 million
2	0.737	0.86 million
4	1.073	0.59 million

Individual kernel run times were obtained on a single Tesla K20c to eliminate potential sources of error, as shown in Table 6.3. It is clear that the reconstructive Fourier transform step, labeled finalFFT, consumed the largest percentage of time during the processing of the sample.

A common method of measuring software quality is through the use of software metrics. Coupling is one measure of software quality. Parameter coupling can be defined as a measure of the number of method calls between classes [29]. There

Table 6.3
Kernel run times for the sample data set 832×240×240×11 and an FFT size of 16,384 on a single Tesla K20c utilizing HyperQ with 16 streams. System memory transfer times are not shown for cast_to_float and normalize kernels as they are normally hidden as a result of pipelining.

Kernel	**Time (ms)**	**Bandwidth (GB/s)**
cast_to_float	24.241	121.519
Hann_filter	21.292	184.468
transpose	31.460	133.168
reduce	54.395	39.111
subtractDC	29.910	141.160
FFT	159.231	133.766
iFFT	381.751	75.976
k-space	354.982	81.705
finalFFT	1975.260	78.312
intensity	88.196	60.377
minmax	15.093	88.203
normalize	22.292	74.650

are three points of coupling between the GPU processing code and the Bloomington control code: a call to the constructor for the cuAOOCT object, an explicit call to the cuAOOCT *run()* method, and an implicit call to the cuAOOCT destructor. Data coupling exists because B-scans are packaged onto objects that are passed to the cuAOOCT *run()* method. With loose coupling often comes high cohesion. Cohesion is the measure of relatedness between module components. High cohesion exists if module components are not divisible. In fact, by design and the data dependency that must exist within the cuAOOCT subclasses to process AO-OCT data, the methods of the classes are functionally cohesive and sequential cohesion exists between each of the processing steps. This level of cohesion allows increased kernel reusability for future processing strategies [30, 31].

7 SUMMARY

The near linear reduction in computation time with an increase in core count is promising for future hardware improvements. By virtue of the way the code is designed, the AO-OCT computation will scale automatically with the GPU resources it detects, and balance the workload between GPUs. Although having a final FFT of 4096 allows better than real-time results at over one million A-scans per second, there may exist a set of GPUs in a year or two that might be able to compute with a scale factor of four in real-time. This would allow a final FFT of 16,384 complex numbers.

One point of future work that should be investigated is batch processing of multiple B-scans at a time. Others have shown improved throughput with such a set-up [5]. The main bottleneck in the current approach is the large FFT size and the small number of A-scans per batch. Current batches are only the size of the number of A-scans per B-scan. Larger FFT batches should increase throughput from the CUDA FFT library. Due to loose coupling and the use of the strategy design pattern, such a class could extend the CuAOOCT class and be added with minimal difficulty.

As speed and accuracy increase, so does the clinical value of AO-OCT. The approach proposed and implemented will allow for future growth and can be easily modified or extended using the principles of object oriented programming.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Kang Zhang and Jin U Kang. Graphics processing unit-based ultrahigh speed real-time fourier domain optical coherence tomography. *Selected Topics in Quantum Electronics, IEEE Journal of*, 18(4):1270–1279, 2012.

[2] Brandon A Shafer, Jeffery E Kriske, Omer P Kocaoglu, Timothy L Turner, Zhuolin Liu, John Jaehwan Lee, and Donald T Miller. Adaptive-optics optical coherence tomography processing using a graphics processing unit. In *Engineering in Medicine and Biology Society (EMBC), 2014 36th Annual International Conference of the IEEE*, pages 3877–3880. IEEE, 2014.

[3] Fftw home page. http://fftw.org/. (Visited on 11/11/2014).

[4] Yuuki Watanabe and Toshiki Itagaki. Real-time display on fourier domain optical coherence tomography system using a graphics processing unit. *Journal of Biomedical Optics*, 14(6):060506–060506, 2009.

[5] Yifan Jian, Kevin Wong, and Marinko 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*, 18(2):026002–026002, 2013.

[6] Kang Zhang and Jin U Kang. Real-time intraoperative 4d full-range fd-oct based on the dual graphics processing units architecture for microsurgery guidance. *Biomedical Optics Express*, 2(4):764–770, 2011.

[7] David Huang, Eric A Swanson, Charles P Lin, Joel S Schuman, William G Stinson, Warren Chang, Michael R Hee, Thomas Flotte, Kenton Gregory, Carmen A Puliafito, et al. Optical coherence tomography. *Science*, 254(5035):1178–1181, 1991.

[8] Joel S Schuman. Spectral domain optical coherence tomography for glaucoma (an aos thesis). *Transactions of the American Ophthalmological Society*, 106:426, 2008.

[9] John Rogers, Adrian Podoleanu, George Dobre, David Jackson, and Frederick Fitzke. Topography and volume measurements of the optic nerve using en-face optical coherence tomography. *Optics Express*, 9(10):533–545, 2001.

[10] Omer P Kocaoglu, Timothy L Turner, Zhuolin Liu, and Donald T Miller. Adaptive optics optical coherence tomography at 1 mhz. *Biomedical Optics Express*, 5(12):4186–4200, 2014.

[11] RB Blackman and JW Tukey. Particular pairs of windows. *The Measurement of Power Spectra, From the Point of View of Communications Engineering*, pages 95–101, 1959.

[12] Murtaza Ali and Renuka Parlapalli. Signal processing overview of optical coherence tomography systems for medical imaging. *Texas Instruments Inc. Application Note (SPRABB9)*, 2010.

[13] Barry Cense, Nader Nassif, Teresa Chen, Mark Pierce, Seok-Hyun Yun, B Park, Brett Bouma, Guillermo Tearney, and Johannes de Boer. Ultrahigh-resolution high-speed retinal imaging using spectral-domain optical coherence tomography. *Optics Express*, 12(11):2435–2447, 2004.

[14] NVIDIA. cufft - cuda toolkit documentation. http://docs.nvidia.com/cuda/cufft, August 2014. (Visited on 10/03/2014).

[15] Gaozhi Xiao and Wojtek J Bock. *Photonic Sensing: Principles and Applications for Safety and Security Monitoring*, volume 227. John Wiley & Sons, 2012.

[16] Milton Abramowitz and Irene A Stegun. *Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables*. Number 55. Courier Dover Publications, 1972.

[17] M. Hollins. *Medical Physics*. Bath Advanced Science Series. Nelson Thornes, 2001.

[18] Lelia Adelina Paunescu, Daniel X Hammer, R Daniel Ferguson, Siobahn Beaton, Hiroshi Ishikawa, John C Magill, Gadi Wollstein, and Joel S Schuman. Active retinal tracker for clinical optical coherence tomography systems. *Journal of Biomedical Optics*, 10(2):024038–02403811, 2005.

[19] Charles J Murray and Arthur L Norberg. *The Supermen: The Story of Seymour Cray and the Technical Wizards Behind the Supercomputer*. Wiley, 1997.

[20] David E Womble, Sudip S Dosanjh, Bruce Hendrickson, Michael A Heroux, Steve J Plimpton, James L Tomkins, and David S Greenberg. Massively parallel computing: A sandia perspective. *Parallel Computing*, 25(13):1853–1876, 1999.

[21] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, 2013.

[22] High-performance computing (hpc) in a real-time environment - national instruments. http://www.ni.com/white-paper/7431/en/. (Visited on 11/24/2014).

[23] NVIDIA. Nvidia kepler gk110 architecture whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012. (Visited on 11/11/2014).

[24] NVIDIA. Programming guide - cuda toolkit documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#intrinsic-functions, 2014.

[25] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing matlab through just-in-time specialization. In *Compiler Construction*, pages 46–65. Springer, 2010.

[26] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[28] NVIDIA. An efficient matrix transpose in cuda c/c++ - parallel forall. http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/. (Visited on 10/28/2014).

[29] Jeff Offutt, Aynur Abdurazik, and Stephen R Schach. Quantitatively measuring object-oriented couplings. *Software Quality Journal*, 16(4):489–512, 2008.

[30] James M Bieman and Linda M Ott. Measuring functional cohesion. *Software Engineering, IEEE Transactions on*, 20(8):644–657, 1994.

[31] James M Bieman and Byung-Kyoo Kang. Measuring design-level cohesion. *Software Engineering, IEEE Transactions on*, 24(2):111–124, 1998.