HHS Public Access

Author manuscript

Conf Proc IEEE Eng Med Biol Soc. Author manuscript; available in PMC 2015 August 29.

Published in final edited form as:

Conf Proc IEEE Eng Med Biol Soc. 2014 ; 2014: 3877–3880. doi:10.1109/EMBC.2014.6944470.

Adaptive-optics Optical Coherence Tomography Processing Using a Graphics Processing Unit*

Brandon A. Shafer[1], **Jeffery E. Kriske Jr**[2], **Omer P. Kocaoglu**[3], **Timothy L. Turner**[3], **Zhuolin Liu**[3], **John Jaehwan Lee**[1], and **Donald T. Miller**[3]

[1]Dept. of Electrical and Computer Engineering, Indiana University Purdue University at Indianapolis, Indianapolis, IN 46202 b.a.shafer@ieee.org

[2]Dept. of Computer Science, Indiana University Purdue University at Indianapolis, Indianapolis, IN 46202

[3]School of Optometry, Indiana University, Bloomington, IN 47405

Abstract

Graphics processing units are increasingly being used for scientific computing for their powerful parallel processing abilities, and moderate price compared to super computers and computing grids. In this paper we have used a general purpose graphics processing unit to process adaptive-optics optical coherence tomography (AOOCT) images in real time. Increasing the processing speed of AOOCT is an essential step in moving the super high resolution technology closer to clinical viability.

I. 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. Adaptive-optics has even allowed for the discovery of new properties of cone photoreceptors [1], [2]. As with a lot of medical research, the end goal of technological advances of this nature are to aid in clinical diagnosis and treatment of patients. However, there exists in clinical settings 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

*This study was supported by Indiana University Collaborative Research Grant fund of the Office of the Vice President for Research, and National Eye Institute grants R01-EY018339 and P30-EY019008.

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.

The way that graphics processors work is different from a conventional CPU such as the Intel or AMD processors in personal computers. CPUs are now very powerful devices that can handle numerous sequential instructions (step-by-step commands) and work at high clock rates. 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 production in AOOCT, in real time. While some research has been done recently in using GPUs for OCT processing [3], [4], it is still a fairly new research area and as far as we know, this is the first it is being applied to ultrahigh resolution AOOCT for use in the human retina.

II. AOOCT

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 ultrahigh resolution spectral domain coherence tomography with adaptive-optics, researchers can get 3D resolutions as fine as $3 \times 3 \times 3\ \mu m^3$ [5], [6].

The evolution of this technology started with optical coherence tomography (OCT) [7], first published with use *in vivo* in 1993 [8], [9]. 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" [7].

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 [6].

To summarize, optical coherence tomography exploits the fact that different tissues reflect light differently in order to create a high resolution image. Adaptive-optics then corrects for diffraction caused by the eye's lens. 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 becomes useful for a clinician. In talking about AOOCT images in this paper, the terms A-line, B-scan, and volumes are used as shown in Figure 1.

III. GPU Processing

Performing the calculations of the AOOCT process on a GPU has several challenges. The ultimate goal of utilizing a GPU is to make the processing time real-time; that is, the images are processed as quick or quicker than they are acquired.

A. Parallelization

The GPU performs calculations a B-scan at a time. Algorithm 1 outlines the process that an individual B-scan goes through. Each step in the algorithm is performed on the GPU as a kernel (function that runs on the GPU) and it was necessary to optimize each step by itself and in relation to the whole process, using the structure of the GPU to greatest advantage.

1) Convert to float—Line 2 of Algorithm 1 converts the incoming data from 16-bit unsigned integers to 32-bit floats. To perform the calculations necessary for reconstruction, first data needs to be copied to the GPU from CPU RAM and converted to float. When our program is initialized, memory in the RAM is registered as pinned memory with the GPU to facilitate what is called zero copy. This enables faster copies and allows the kernel to reference the data directly without an extra memory copy. Each pixel of data is cast from an unsigned 16-bit integer to a single precision 32-bit float and stored in global memory on the GPU.

2) Subtract DC component—Line 3 from Algorithm 1 is to remove the DC component of the incoming A-line signals for the entire B-scan. 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 shown in Equation 1.

$$\bar{A}_i = \sum_j A_{ij} \quad (1)$$

where i is an element of A-line and j is the place in B-scan. Then the averaged component is subtracted from each element in the B-scan as shown in Equation 2.

$$A\text{new}_{ij} = A_{ij} - \bar{A}_i \quad (2)$$

When first approaching this problem, the most straight-forward 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.

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 [10]. Generally, this allows for reducing the number of steps to $\log_2 N$ per warp (a set of threads). 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 (i.e., scattered) 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.

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 [11] shows how to use tiling and shared memory as well as diagonal block reordering in order to facilitate memory coalescing and prevent memory bank conflicts. Our transpose makes use of all of these techniques (tiling, etc.) to optimize our speed.

Algorithm 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 unsigned integer to float
3: Subtract DC component from B-scan
4: Pad A-lines
5: Align to *k*-space
6: Compensate for dispersion
7: Do 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**

Once the B-scan is transposed, the reduction calculates the total along each row and stores the calculated average in a new vector. After this step is complete, another kernel is launched that subtracts the average from each element.

3) Pad A-lines—The next step, Line 4 of Algorithm 1, is padding A-lines. To better facilitate processing on the GPU and to create better images, the input signals are re-sampled through a four step process. First, the initial signal is padded on both sides of the A-line to have a size of a power of two to better facilitate the GPU. Second, the signal is passed through an FFT. 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.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i 2\pi k n / N} \quad (3)$$

Third, each A-line is padded with zeros. If the full A-line was seen, the padding would be in the middle, but since we are using hermitian symmetry, only one side of the A-line is stored and padding only needs to be applied to the end. 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 $N_{padded}/2 + 1$. Fourth, an inverse FFT is performed as follows.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{i2\pi kn/N} \quad (4)$$

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.

4) Align to k-space—Line 5 of Algorithm 1 is to align the A-lines to *k*-space. When captured, the initial images happen to have unevenly spaced wavelengths; in order to create an evenly spaced final image, the A-line spectra must be realigned from λ-space to *k*-space, where $k = 2\pi/\lambda$. Additionally 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 wavelength are converted to *k*-space as in Equation 5 and Equation 6.

$$k_{\min} = (2 \cdot \pi)/\text{wavelength}[0] \quad (5)$$

$$k_{\max} = (2 \cdot \pi)/\text{wavelength}[\text{end}] \quad (6)$$

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 are converted to λ-space so that the current A-line spectra can be interpolated to the new evenly spaced values.

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.

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. 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 $\lambda_{left}i$ which is the value that is less than or equal to the new wavelength.

During the actual interpolation kernel, Equation 7 is performed to find the new A-line spectra.

$$A_i^{'}=A_{\text{left}i}+(A_{\text{right}i}-A_{\text{left}i})\frac{\lambda_i^{'}-\lambda_{\text{left}i}}{\lambda_{\text{right}i}-\lambda_{\text{left}i}} \quad (7)$$

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. The interpolation described in Equation 7 is performed each A-line, while the mapping is only calculated on the first A-line and stored for all the rest.

5) Compensate for dispersion—OCT images are often blurred from an optical effect called dispersion [12], [13]. The light that enters the eye is dispersed because of the refractive index of the tissue. The dispersion affects different wavelengths differently. According to Cense et al., "the relation between phase θ(*k*) and the multiple orders of dispersion can best be described by a Taylor series expansion:

$$\theta(k)=\theta(k_0)+\frac{\partial\theta(k)}{\partial k}\Big|_{k_0}(k_0-k)+\frac{1}{2}\cdot\frac{\partial^2\theta(k)}{\partial k^2}\Big|_{k_0}(k_0-k)^2+\ldots+\frac{1}{n!}\cdot\frac{\partial^n\theta(k)}{\partial k^n}\Big|_{k_0}(k_0-k)^n \quad (8)$$

with λ_0 the center wavelength and k_0 equal to $2\pi/\lambda_0$" [13].

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 from a calibration file. The complex phase terms are found in the human eye using a well-reflecting reference point, the center of the fovea. The resulting B-scans are complex-valued. In terms of CUDA, this is a very simple kernel that reads in B-scans and multiplies by the complex coefficients provided from calibration.

6) Process 1D Fast Fourier Transform—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.

7) Perform final conversion to image—At this point, the B-scan is reconstructed into a complex-valued image. To enable viewing by a clinician, several steps can be performed to make it more palatable. The following steps are highly depended on the preference of the end user, but are still done on the GPU if desired and are straightforward to implement.

1. Crop image.
2. Convert complex numbers to intensity.
3. (Optional) Convert to log scale.
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 with Equation 9.

$$I_{ij}=\Re(A_{ij})^2+\Im(A_{ij})^2 \quad (9)$$

B. Fast Fourier Transform (FFT)

For FFTs, we turned to cuFFT, a library initially based on FFTw for C++, but developed and optimized for running on GPUs [14]. We utilize three FFTs in our algorithm: a real-to-complex FFT, a complex-to-real inverse FFT, and a complex-to-complex final FFT. The library allows us to use advanced memory layout and batched processing. The advanced memory layout allows us to place the output of the first FFT in the zero-pad process directly into the memory for the beginning of the IFFT. The memory can go from size N real values to $N/2 + 1$ complex values to $M/2 + 1$ complex values back to M real values, and the advanced memory layout makes that easier. It also takes advantage of hermitian symmetry to save on storage space, and the reduced size makes the resulting actions more efficient and faster.

C. Data Flow

The steps are pipelined to enable better utilization of GPU resources. CUDA allows for separation of kernels into concurrent streams. Each stream can run independently from each other. Generally speaking, the busier the GPU is, the higher our throughput is going to be. As a B-scan comes into the GPU, all of the steps taken on that B-scan as described in Algorithm 1 are performed on a single stream. The next B-scan is assigned to the next stream.

In order to pipeline the data, we needed separate data paths for each stream. By data paths we mean the memory space allocated for data to be read from and written to by each processing step, e.g., the memory the initial FFT writes to and the inverse FFT reads from. Each stream needs to have its own allocation of memory. The streams could access any of the global memory, but trying to coordinate memory accesses between streams would be difficult and would likely slow the whole operation. When the global memory is plentiful enough, creating separate memory for each stream is much simpler and easier to manage.

The cuFFT library relies on making a plan for an FFT. We utilize three different plans for the 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 does allocate 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 [14].

IV. Results and Future Work

The end result is a GPU program that speeds up AOOCT image processing to about 32× as compared to what can be done on a CPU using C++ and 1945× as compared to the original

Matlab version. On an image set with an A-line width of 832 pixels, 240 A-lines per B-scan, 240 B-scans per volume, and a total of 11 volumes, our GPU implementation computes in 1450 milliseconds compared to 47026 milliseconds in the CPU implementation and over 47 minutes in original Matlab implementation. The acquisition speed for that data is 2.534 seconds, so the calculation time is more than sufficient for now. In future work, as acquisition speeds are increased, the GPU may need to be expanded to using multiple GPUs for calculation.

References

1. Pallikaris A, Williams DR, Hofer H. The reflectance of single cones in the living human eye. Investigative Ophthalmology & Visual Science. 2003; 44(10):4580–4592. [PubMed: 14507907]
2. Rha J, et al. Rapid fluctuation in the reflectance of single cones and its dependence on photopigment bleaching. ARVO Meeting Abstracts. 2005; 46(5):3546.
3. Jian Y, Wong K, Sarunic MV. Graphics processing unit accelerated optical coherence tomography processing at megahertz axial scan rate and high resolution video rate volumetric rendering. Journal of Biomedical Optics. 2013; 18(2):26 002–26 002.
4. Zhang K, Kang JU. Graphics processing unit-based ultrahigh speed real-time fourier domain optical coherence tomography. IEEE J. Sel. Topics. Quantum Electron. 2012 Jul; 18(4):1270–1279.
5. Zawadzki RJ, et al. Ultrahigh-resolution adaptive optics - optical coherence tomography: toward isotropic 3 μm resolution for in vivo retinal imaging. Proc. SPIE. 2007; 6429 pp. 642 909–642 909–9.
6. Miller DT, et al. Adaptive optics and the eye (super resolution OCT). Eye. 2011 Mar; 25(3):321–330. [PubMed: 21390066]
7. Huang D, et al. Optical coherence tomography. Science. 1991 Nov.254(5035):1178. [PubMed: 1957169]
8. Fercher AF, et al. In vivo optical coherence tomography. American Journal of Ophthalmology. 1993; 116(1):113–114. [PubMed: 8328536]
9. Swanson EA, et al. In vivo retinal imaging by optical coherence tomography. Opt. Lett. 1993 Nov; 18(21):1864–1866. [PubMed: 19829430]
10. Harris M. Optimizing parallel reduction in CUDA. http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.
11. Ruetsh G, Micikevicius P. Optimizing matrix transpose in CUDA. 2009 Jan. http://www.cs.colostate.edu/cs675/MatrixTranspose.pdf.
12. Zhang Y, et al. Adaptive optics parallel spectral domain optical coherence tomography for imaging the living retina. Opt. Express. 2005 Jun; 13(12):4792–4811. [PubMed: 19495398]
13. Cense B, et al. Ultrahigh-resolution high-speed retinal imaging using spectral-domain optical coherence tomography. Opt. Express. 2004 May; 12(11):2435–2447. [PubMed: 19475080]
14. CUDA toolkit documentation: CUFFT. 2013 Aug. http://docs.nvidia.com/cuda/cufft/index.html.

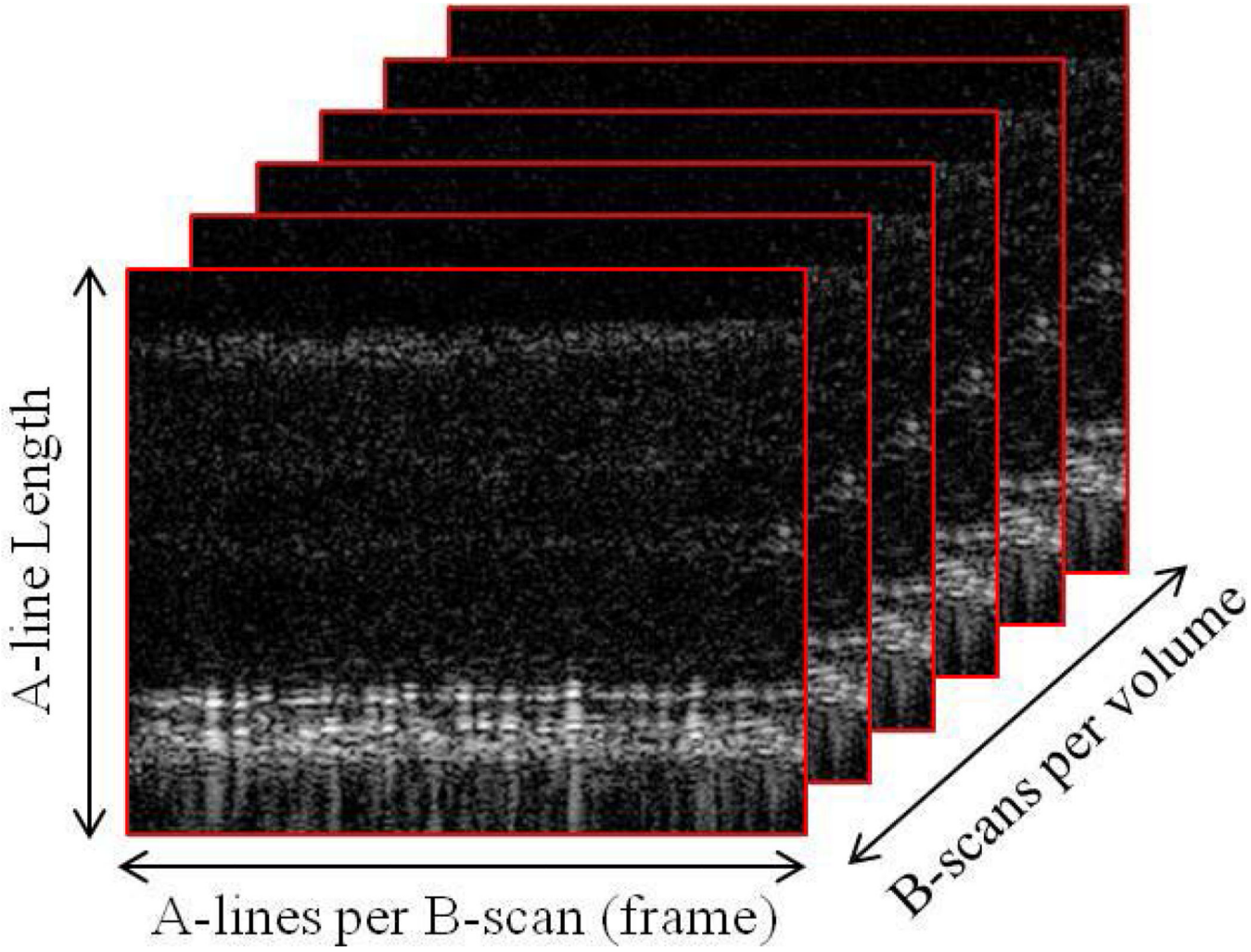

Fig. 1.
Volume AOOCT images.