Acceleration of Spectral Domain Phase Microscopy for Optical Coherence Tomography

Brian Frost and Xuanyi Liao
Electrical Engineering
Columbia University
Email: bf2458@columbia.edu, xl2875@columbia.edu

***Abstract*—Spectral domain optical coherence tomography (SD-OCT) is an imaging modality which is used within biomedical research. SD-OCT allows for depth imaging of samples, and can be used in conjunction with a technique known as spectral domain phase microscopy (SDPM) to measure vibrations with sub-nanometer sensitivity. SDPM and OCT can measure vibrations at a depth into a sample, which has made it a useful tool in measuring vibrations within the cochlea *in vivo*. SDPM requires a series of SD-OCT measurements to be taken at a location over a long period of time at a relatively high sampling rate to achieve a high signal-to-noise ratio (SNR). This incurs a large data burden which can cause processing to dominate the experimentation time. We have used the NVIDIA CUDA language to parallelize the processing of these large SD-OCT data sets, achieving very large speed-ups in critical time-consuming portions of the processing pipeline. We have tested this parallel method on *in vivo* data, and have validated it against an established MATLAB-based serial processing algorithm.**

I. Introduction

Our goal is to use the NVIDIA CUDA language to parallelize the processing of large SD-OCT data sets to obtain vibration information through SDPM, incurring a large speed-up over the currently used serial algorithms. In this introduction, we will expose the mechanisms of SD-OCT, SDPM and in doing so, the processing pipeline which we are looking to accelerate.

A. Spectral Domain Optical Coherence Tomography

Optical coherence tomography (OCT) was initially implemented through what is known as "time domain OCT" (TD-OCT), and while these machines are no longer in use, they provide an easier way through which to understand the mechanisms of OCT. Figure 1 shows the schematic of a TD-OCT system.

The light source used is a low-coherence light source, and the optical system architecture is that of a Michelson-Morley interferometer (see [1]). As the reference mirror scans, the optical path difference between the reference arm and the sample arm of light changes. When these optical path length differences are within a coherence length of one another, interference fringes will be seen at the detector. If the sample is made up of multiple reflectors at varying depths, then interference fringes will appear at multiple reference mirror distances and a reflectivity profile of the sample can be built.

Mathematically, if the input signal has electric field amplitude given by $E_0(x)$, the reference and sample signal differ by some optical path length difference only, and thus at the detector a field with amplitude proportional to $E_0(x+\tau_s) + E_0(x+\tau_r)$ is seen. As the reference varies, so too does τ, and the detector integrates the intensity (proportional to the square of the electric field). That is to say the detected signal is proportional to

$$\int_0^L (E_0(x+\tau_s) + E_0(x+\tau_r))^2 \, d\tau_r.$$

Expanding this square and distributing the integral, we get two fixed field intensities of the form $\int E_0$, and one term proportional to

$$f(x) = \int_0^L E_0(x+\tau_s)E_0(x+\tau_r)\, d\tau_r. \qquad (1)$$

The information we desire in the A-Scan is encoded in this term, which is an autocorrelation of the input signal. The true desired term – the electric field magnitude, and thus reflectivity, in the sample arm as a function of displacement – requires a deconvolution of this autocorrelation with the background signal.

In short, OCT uses the fringe visibility profile of a sample to determine the reflectivity profile of a substance in the dimension of depth into a sample. In TD-OCT, a single scanning period of the reference mirror gives a one-dimensional line of data into the sample known as an A-Scan. Here, the resolution is largely determined by the coherence length of the lght source. To gain two- or three-dimensional images, A-Scans are taken at a number of points along the sample and then displayed alongside one another.

SD-OCT removes the necessity for a scanning mirror, but uses the same Michelson-Morley architecture. Figure 2 shows an SD-OCT system schematic. There are only a few differences – the reference mirror does not scan, and the detector has been replaced by a diffraction grating and a spectrometer. This system is able to avoid the use of a scanning mirror by obtaining the fringe visibility profile indirectly.

As stated above, the information required to form an A-Scan is in the autocorrelation function shown in Equation 1. The Wiener-Khinchin theorem states that the power spectrum, or the intensity in *frequency*, is related to the autocorrelation by the Fourier transform. That is to say, should we collect the power spectrum, we can then compute the autocorrelation to obtain our A-Scan (see [1]).

The diffraction grating or prism separates our superposition signal into its wavelength components (i.e. it's frequency components), and the spectrometer measures the intensity of the light at each frequency. Thus, we can instantaneously measure something akin to the power spectrum and compute the autocorrelation and subsequent deconvolution to gain our reflectivity profile.

However, this is not quite the power spectrum, so the post-processing is not so simple as "taking a Fourier transform". The reflectivity profile we desire is in space, and the Fourier transform pair of space is spatial frequency or wavenumber, *not* wavelength. However, wavenumber and wavelength space are directly related by the relation $k = 2\pi/\lambda$, where k is the wavenumber is λ is the wavelength.

Furthermore, we receive a wavelength-domain signal for the *entire* superposition signal. We know that we are actually not interested in the "background" component of this signal, much like in the TD-OCT where we are not interested in the unchanging intensity of the reference signal. We must subtract this out at the beginning of our processing pipeline.

The benefits of SD-OCT are clear – not having to make move a scanning mirror mechanically to take a single measurement yields a faster, more stable method of arriving at the same result. However, more processing is required to achieve the reflectivity profile. Fourier transforms are well-understood, well-optimized objects, but for very large data sets they still take non-negligible time. Similarly, for large data sets, the transformation from wavenumber to wavelength space takes non-negligible time.

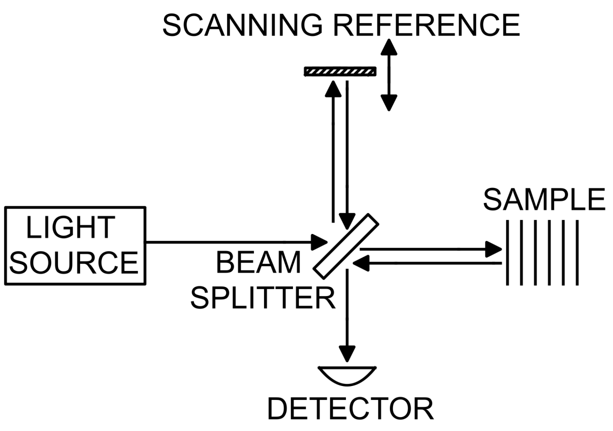

Fig. 1. A schematic of a time domain OCT system imaging a sample containing a number of evenly spaced reflectors.

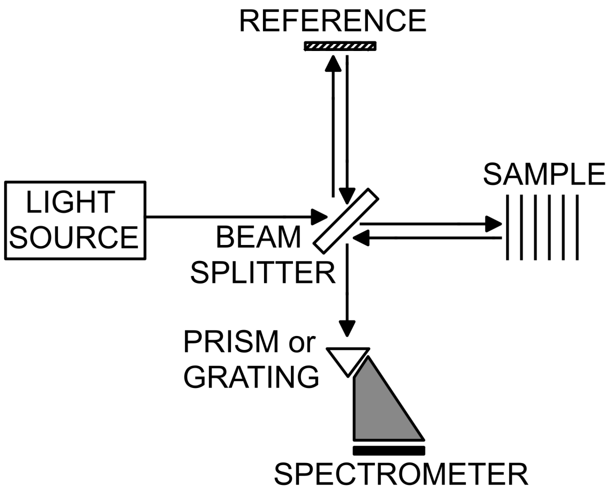

Fig. 2. A schematic of a spectral domain OCT system imaging a sample containing a number of evenly spaced reflectors.

B. Spectral Domain Phase Microscopy

A-Scans can be captured in SD-OCT as fast as the spectrometer can be clocked, which can vary by machine (in our case, 100 kHz) – this speed is known as the linerate. SD-OCT systems have far faster linerates than TD-OCT systems, as they are able to capture entire A-Scans in one instance. This high linerate lends itself to vibration measurements, as for a system with a linerate f_s, Nyquist's theorem states that vibrations up to $f_s/2$ can be measured.

We must consider the limitations of these vibration measurements, however – a naïve method would require large enough vibrations within the dimension of the A-scan that the reflectivity profile would change by more than the system's resolution. This resolution, determined by the coherence length, is not usually under 1 μm, and vibrations within the cochlea are on the order of nanometers.

SDPM allows for displacement measurements at super-resolution levels, with sensitivity below one nanometer, by using the phase of the A-Scan rather than the magnitude. In SD-OCT, we retrieve the reflectivity profile through a Fourier transform of the power spectrum as described above. The magnitude of this Fourier transform in particular is what gives the A-Scan obtained through TD-OCT, by the Wiener-Khinchin theorem. However, the phase of this Fourier transform, for sub-pixel structures, encodes the position within a given pixel at which the structure lies. That is, should an object be smaller than is necessary to resolve with the OCT system, its displacement within the pixel it falls in can be tracked through its phase in time.

For measurements in which nanometer-scale vibrations are occurring, the magnitude of the A-Scan will not change whatsoever, but the phase at vibrating points will be directly proportional to the displacement. In fact,

$$x(t) = \frac{\phi(t)\lambda}{4\pi n}, \tag{2}$$

where x is displacement, phi is phase, λ is the center wavelength of the light source and n is the index of refraction. This, note, is computed *per pixel*.

A derivation of this fact, as well as a more comprehensive treatment of OCT can be found in [2].

II. Data Characteristics

It is important to have an understanding of the form of the data we are processing. In this section we will describe the devices used to create the data we are working with, as well as the behavior of the structure we are imaging.

A. OCT System

We use a Thorlabs Telesto SD-OCT system with center wavelength 1,310 nm. The spectrometer is 2,048 pixels, and the data corresponding to each A-Scan is given as an integer intensity detected at each pixel (i.e. at each wavelength). The system has a maximum linerate of 100 kHz, and is clocked by a Tucker-Davis Technologies analog-to-digital system (TDT) at 97,656.25 Hz. This strange linerate is a feature of the TDT system. The pixel size in depth is about 2.2 μm, but the resolution of the system is far larger – about 11 μm.

B. Anatomy

We take data *in vivo* from anesthetized gerbils. Figure 3 shows a cross-section of the cochlea – the main structure of the inner ear. The cochlea spirals about from its widest point, the base, to its narrowest point, the apex. We take A-Scans through the round window membrane – a relatively transparent membrane in the basal region of the cochlea. Through this round window, we wish to see the hearing organ, or the organ of Corti (OOC), as it vibrates with respect to sound stimulus.

Figure 4 shows both a two-dimensional and one-dimensional scan acquired from a gerbil with our OCT system. The two-dimensional scan is referred to as a B-Scan, and is formed by taking a number of A-Scans and displaying them side-by-side. Using this B-Scan, we are able to find the structures of interest, and determine a line of interest through which to take an A-Scan. Such an A-Scan is shown in this same figure. Peaks in the A-Scan correspond to anatomical structures, and they are usually the points we which to measure vibration at.

C. Data Collection

We stimulate the gerbil ear using a sound stimulus which is the superposition of multiple tones. We collect A-Scans at one position in the basal region of the gerbil's cochlea. The cochlea encodes frequency spatially, in that regions of the cochlea vibrate more in response to certain frequencies than others. The region we view responds most at a "best frequency" of about 22 kHz, and our stimuli have components between about 1 kHz and 40 kHz. These all fall below the Nyquist rate of our system, which is about 50 kHz.

We present this sound stimulus and record A-scans at the linerate determined by the TDT. We then fetch this data from the memory onboard the OCT system, in the form of a RAW file. This is a binary file, stored as an array of 16-bit unsigned integers. Every 2,048 consecutive integers is one A-Scan. The first 324 A-Scans are what are called "background scans" – they are scans in which only the reference arm intensity is measured. We fetch the data using a simple C++ program written using a library supplied by Thorlabs known as the "Spectral Radar Software Development Kit".

We repeat this process using the same stimulus at several volumes. The data we use in this experiment used a 40-frequency stimulus played for about 5 seconds at each of 50, 60, 70 and 80 decibel sound pressure levels (dB SPL). Note that each amplitude corresponds to its own raw file. The nmber of recordings taken is chosen to be a power of 2, so 524,288 recordings are taken rather than the slightly smaller required number for exactly 10 seconds of data. We also must note that the first and last several scans taken of the sample are inherently noisier than the rest, due to initial mechanical transients in mirror motion. Due to this, we actually take 2,048 extra scans and ignore the first and last 1,024 scans in the data set.

So for each sound pressure level, we have 324 background scans, 524,288 scans of interest and 2048 noisy scans we do not use, constituting 526,660 scans total. Each scan consists of 2,048 2-byte integer, meaning each RAW file slightly more than 2 GB. The 16-bit integers are a representation of the power received at each pixel in the line camera on the Telesto system. The Thorlabs Telesto manual states that these values are off by a constant factor of 550 from the actual power received for hardware-specific reasons. Thus, the true received power is the input multiplied pointwise by 550.

III. Methods

To understand the regions of the problem which can be parallelized, we must first understand the serial pipeline required to obtain the desired data from the RAW input. In this section, we present the serial pipeline, an analysis of the parallelizability of each step, and methods by which each step is tested for optimal parallel performance.

A. The Pipeline

1) Loading Data into RAM: Although uneventful, the loading of the RAW file from disk to RAM is a major bottleneck for the performance of the processing – on the computer hosting the OCT system, loading in data can take about 5-25 seconds per RAW file. While in the serial algorithm, this does not dominate the computation time, in the parallel algorithm we hope that this is the lengthiest step.

Although mathematically uninteresting, the scaling by 550 must also happen here. Scaling by 550 requires the use of larger integers (32-bit), as this multiplication pushes some inputs above the 16-bit limit causing integer overflow. Thus, recasting more so than multiplying does take a bit of time, and requires 4 GB of RAM in total.

2) Background Fitting: The first 324 scan contain the background. We first average these signals pointwise arithmetically to obtain a single "average background" vector. This average background plays two important roles in processing: it must be subtracted from the information-carrying vectors in the matrix before they are further processed, and it also must be deconvolved from the autocorrelation function in the spatial domain.

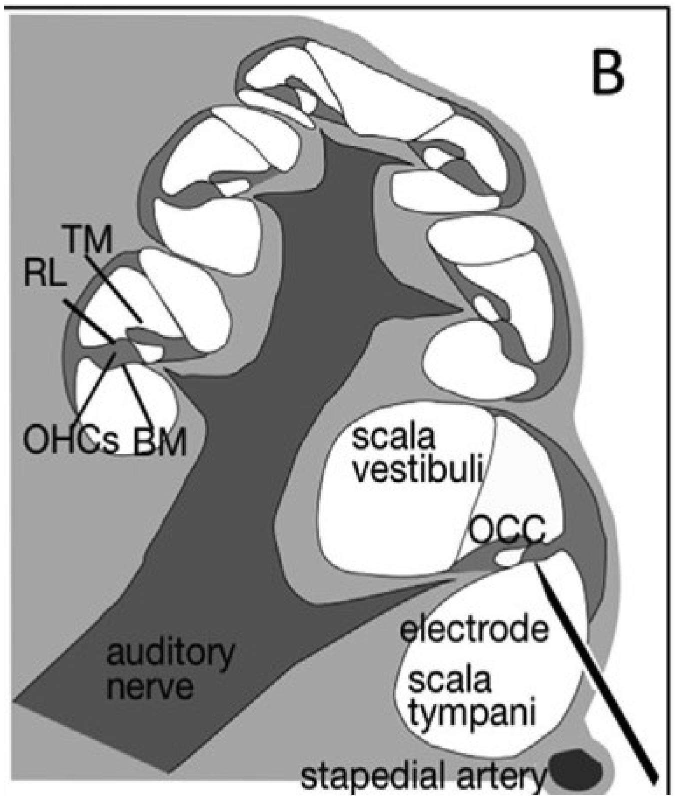

Fig. 3. A cross-sectional schematic of the cochlea, wherein the black arrow on the bottom right gives an estimate of our view through the round window of a basal turn of the cochlea. OCC is organ of Corti complex, meaning the Organ of Corti along with the basilar membrane. TM is tectorial membrane (which we do not see in images), RL is reticular lamina, BM is basilar membrane and OHC is outer hair cells.

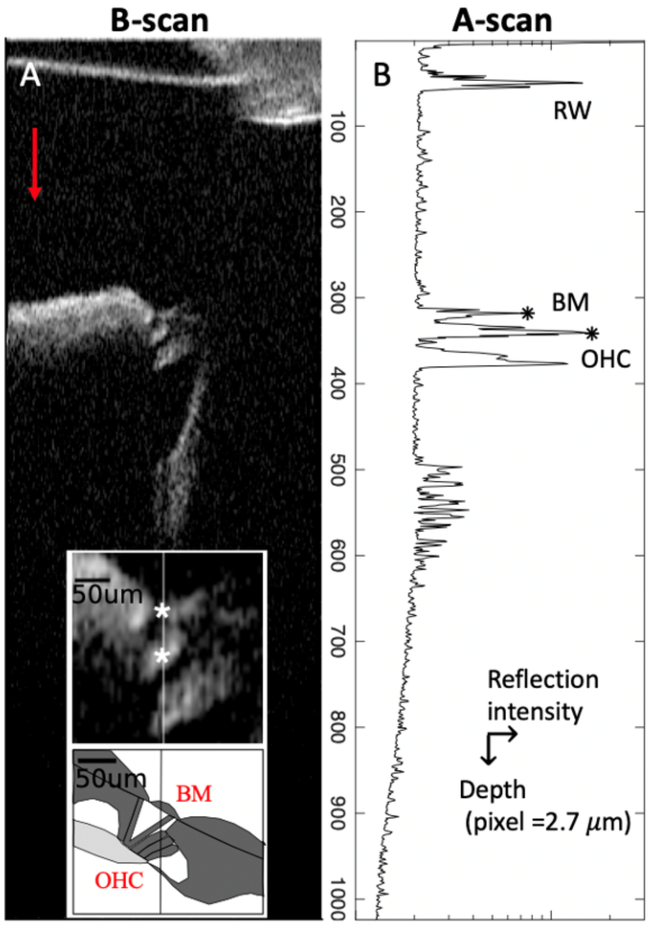

Fig. 4. On the right, a B-Scan or two-dimensional OCT image of the cochlea taken *in vivo*. At the top is the round window membrane, through which we view the inner ear. Towards the middle is the organ of Corti, the hearing organ, wherein we are interested in vibrations. The zoomed in OCT image is compared directly to known anatomy of the inner ear – BM is basilar membrane, OHC is outer hair cells. The line through the center corresponds to the A-Scan or line scan position. We take A-Scans here in time, and the magnitude of such an A-Scan is shown on the right. Peaks in the A-Scan magnitude correspond to the known structures of the ear.

For the former, we are simply performing pointwise subtraction in the time domain. For the latter, we can divide while in the spectral domain rather than deconvolve in the spatial domain. To do this, it is desirable that our signal is smooth, so we must fit the averaged background to some smooth function.

As it has been determined experimentally to work, we choose to fit our averaged background to a 9^{th} order polynomial, and then normalize it so that its highest value is 1. We call this signal the "smooth" background.

3) Window Determination: Whenever a Fourier transform is taken of a signal which has no natural interpretation as being extended periodically (such as our data), we should apply a window other than the naturally implied rectangular window so that the transform-domain representation does not have artificially inflated high-frequency artifacts. We choose a Hanning window, which is pre-computed and simply loaded into memory from a text file at the beginning of the program.

As the background must also be deconvolved from the signal in space, we can combine the Hanning windowing and background deconvolution by building first one "superwindow". This window would be determined by the pointwise division of the Hanning window by the smoothed background in the spectral domain. Then, when it comes time to window the signal, we can window it with the superwindow to perform complete both the windowing and deconvolution tasks at once.

4) Background Subtraction: Each scan must have the averaged background (not smooth background) subtracted from it. This is simply a vector subtraction, but performed over 500,000 times.

5) Superwindowing: Each scan is windowed by the superwindow, which is simply a pointwise multiplication of each scan by the precomputed 2,048-long superwindow vector.

6) Resampling: The data in the wavelength domain must be resampled into the wavenumber domain. This is a bit non-trivial, as evenly spaced wavelengths do not correspond to evenly spaced wavenumbers. Thus, this resampling also requires an interpolation.

A linear algebraic approach simplifies this problem a great deal. The wavelength domain vector is a 2,048-vector represented in one basis, where as the wavenumber vector is simply a representation of this same data in a separate basis of size

2,048. Thus, there exists a 2,048 $\times$ 2,048 change-of-basis matrix which maps all wavelength domain representations to wavenumber domain representations. This change-of-basis matrix is pre-computed and loaded in from a text file at the start of the program. Resampling is performed by multiplying each scan, individually, by this matrix.

Within the serial algorithm, this is the step which takes the most time – upwards of 20 minutes.

7) Fourier Transform: A Fourier transform is taken scan-wise to obtain the A-Scans in depth. The output of this transform is about 500,000 2,048-long complex A-Scans, from which we care only about the phase. However, it is important to note that this construct is twice as large as the input: about 4 GB.

As we can see from the B-Scan in Figure 4, we only need vibration measurements at a small subset of the total number of pixels in a line. That is, less than 100 of the 2,048 pixels in each line actually correspond to anatomical structures within the cochlea. Thus, we can crop these A-Scans to be significantly smaller – smaller, even, than the input data.

8) Spectral Domain Phase Microscopy: For each pixels in the cropped A-Scan, we actually only care about the phase. We compute the phase using the arctangent of the imaginary part divided by the real part at each point. Once we have computed the phase, we gain the displacement simply through pointwise scalar multiplication as in Equation 2.

B. Parallelizability

1) Background Fitting: Arithmetic averaging can be easily implemented in parallel in a far quicker manner than in serial, as can polynomial fitting and normalization. However, as none of these algorithms are particularly time-consuming in serial for only 324 vectors of size 2,048, it is not immediately clear that the serial algorithm will be so slow as to justify the memory transfers to the device.

Simple serial and parallel algorithms for these steps should be made, and depending on the time required to transfer the background to the GPU when compared to computtion time, we will decide where the computation should be performed.

2) Window Determination: As the superwindow will need to be transferred to the GPU regardless, it is clear that a parallelized creation of the superwindow will be preferable to a serial one. The kernel will simply divide elements of the hanning window by elements of the smoothed background to create the superwindow on the GPU.

3) Background Subtraction and Superwindowing: With the background and superwindow loaded onto the GPU, background subtraction and superwindowing are simple independent scalar subtractions and multiplications applied elementwise. This can be parallelized at arbitrary levels of granularity.

4) Resampling: Resampling, as a matrix multiplication, can naturally be parallelized in a number of ways. However, more interesting is that the change-of-basis matrix is sparse. This opens the door for a number of interesting techniques through which to apply parallelization. Sparse approximations of the matrix of varying sizes, for example, can be tested for accuracy and speed, while certain sizes may allow for the matrix to be stored in shared memory and some may not.

C. Fourier Transform

The Fourier transform has already been optimized for GPUs, and we choose to use a built-in function from a CUDA module to perform the Fourier transform. This will save us time in development, allowing us to explore the impact of other parts of the pipeline deeply.

1) Spectral Domain Phase Microscopy: Much like in the superwindowing and background subtraction steps, this is a set of many simple elementwise operations. Even the arctangent is available as a basic function in CUDA kernels.

IV. Design

We describe here the process by which we determined how to implement each part of the algorithm, and the conclusions we have drawn from experimentation and theory. We touch on important results to do with computation time, but provide more comprehensive details in the following section.

A. Background Fitting

Three steps are required in the background fitting portion of the algorithm – a numerical average, a normalization and a polynomial fitting. To perform normalization, one must first find a maximum value by which to normalize.

As a first test, averaging and maximum finding were implemented in CUDA kernels. For the former, a simple method wherein each thread averaged a single index was applied. For the latter, an efficient reduction algorithm was used. Individually, these kernels performed significantly fast than the serial manifestations of these same algorithms, however the data transfer from host to device dominated the time taken by the algorithms significantly. In fact, the time taken by the data transfer and these two kernels was longer than the time taken to perform the entire background fitting process serially.

Furthermore, a serial implementation of this process in numpy takes only about 0.01 seconds on average, which is very short when compared to even the time required to read the RAW file into RAM. As a result, we choose to implement this algorithm serially.

B. Window Determination

The creation of the window in parallel is actually *also* slower than in serial, as the operation of pointwise division of two 2,048-length vectors is relatively straightforward. However, the bottleneck once again is data transfer. As the window will have to be transferred to the GPU for later computations anyway, it is actually slightly more efficient to perform this operation on the GPU. As this operation takes fractions of a millisecond in either case, this optimization is thoroughly uninteresting.

C. Background Subtraction, Superwindowing and Resampling

We choose to implement all of these items in a single kernel. With the superwindow, background and resampling matrix as inputs, each matrix multiplication operation can operate on the windowed, background subtracted element of the scan rather than performing these three operation on the entire data set in order.The issues of interest are how we choose the block size and how we choose to implement sparse matrix multiplication.

As for the former, we actually cannot load the entire data set onto the GPU at once. While some GPUs have over 2 GB of RAM, we will need more than 4 GB of RAM to compute the Fourier transform later in the pipeline. This is a lot to expect, so we instead split the problem into batches.

Batches are equal-sized sets of scans. Batches are loaded one at a time onto the GPU, and the entire rest of the processing pipeline (background subtraction, superwindowing, resampling, Fourier transform and SDPM) is performed on one batch before the next is handled. That is, batches are traversed serially, but data within each batch is processed in parallel.

Thread blocks in CUDA can contain up to 1,024 threads. Choosing this size for blocks, we can have each thread compute two elements of the resampled scan per scan, and each thread can handle multiple scans. We vary the number of scans in each batch and the number of scans handled by each thread independently.

If the maximum throughput of the GPU is being achieved, than increases and decreases in the batch size will have no measurable effect on computation speed. That is, larger batches will take longer to process, but there will be fewer of them, while smaller batches will take less time but there will be more of them. On top of this, the same total amount of data is being transferred to the GPU across all batches. Thus, so long as the batch size is not made so low so as to underwork the GPU (1 scan per batch, for example), the time taken is approximately fixed.

The same argument applies to the number of scans handled serially by each thread. If the maximum number of parallel tasks is at all times being performed by the GPU, then having more scans per thread leads to more productive, longer lasting threads while fewer scans per thread leads to less productive, shorter lasting threads. So long as each thread does not try to process an enormous portion of the batch so as to clog max throughput, no change is seen in varying the number of scans per thread.

Thus we arrive to the understanding that about 100-200 batches of scans, with each thread processing about 200 scans in the resampling kernel yields the best case scenario, but so too do many values of these two parameters.

Sparse matrix multiplication is a bit more interesting. The change-of-basis matrix is not quite sparse, but only approximately sparse – most elements in it are nonzero but very small compared to the maxima. Furthermore, the most significant values in each row or column are consecutive. We choose to take the N most significant, consecutive values per row, and send an $N\times$2,048 matrix to the device rather than a 2,048$\times$2,048 one. We also must send the indices at which these significant values begin in the original matrix.

On the GPU in use, to fit a matrix of this type in shared memory, we must let N be no larger than 6. This is insufficient for the accuracy of our computations, so we must instead keep the matrix in global memory.

The smallest matrix size that gives sufficiently accurate results is $N = 13$. This means that for each resampled scan element, only thirteen multiplications must be performed rather than 2,048. This is significant, and along with parallelization gives an incredible speed-up.

In our well-established MATLAB algorithm, the resampling step takes the longest time of any step – between 10 and 20 minutes. Using $N = 13$, we find that the background subtraction, superwindowing and resampling step take only about 11 seconds together. This is a two-order-of-magnitude improvement, and places this computation on the same timescale as reading the data to RAM.

D. Fourier Transform

The fast Fourier transform (FFT) is an algorithm for computing the discrete Fourier transform (DFT) of an input vector. This algorithm decomposes a given DFT computation into several smaller DFT computations which in sum are easier to compute. For signals that have length 2^n for some integer n, radix-2 factorization can be used to accelerate these small DFT computations. In our project, each signal is represented as a row vector with length 2,048, so we can benefit from this acceleration.

The FFT is performed after the data is resampled. The resampled data is of the “float” type, but must be converted into the “complex” type before the FFT operation can take place. We use the built-in FFT function in the Cupy library, as it is bold to assume we could write a better FFT algorithm ourselves. The axis of the FFT operation is set to 1, as we wish to perform the FFT row-wise.

E. Spectral Domain Phase Microscopy

SDPM is implemented in a single kernel. It has four inputs and one output. The first input is the result of the FFT, which is a complex matrix with size $BatchSize \times 2,048$; the second input is a constant floating point number $k = \lambda/4\pi n$ which is used to calculate the displacement per pixel; the third input is a cropping factor l which crops the size of each output signal. At the Fowler lab, most data of interest lies in the first 512 pixels, so we let $l = 512$ for our experiments; the last input $BatchSize$ is for boundary condition checking.

For each index in the FFT input, the real and imaginary part of each complex entry are extracted to compute the phase at that pixel. Finally, the phase is multiplied by constant k to calculate the displacement. Because the kernel only involves pointwise operations, theoretically, there will not be any running time difference when using different block sizes.

V. Results

We present here the behavior of all used kernels with respect to variation in parameters such as input size and block size.

These results are used to determine the values which should be used in a practical implementation of this program.

A. *Window Determination*

The superwindow is created through a pointwise division operation of a Hanning window and the smoothed background vector. The Hanning window and smoothed background each have 2,048 points, and the maximum block size for one dimensional blocks in CUDA is 1,024 threads. We test all available block sizes from 1 to 1,024, as shown in Figure 5. It can be concluded that the block size will not affect the speed of this kernel. As a result, we choose 1024 as our block size.

For completeness, we also measure the runtime of the CPU and GPU algorithms while varying the input vector sizes. For this, we use two random input vectors and divide them using either our kernel or numpy pointwise division. As shown in Figure 6, there are no major differences until the input vector reaches a very large size – 2^{22}. Our OCT system has a 2,048-pixel camera, which is why our window is 2,048 in length, however it is worth noting that the GPU algorithm scales to larger camera sizes and thus to different OCT hardware.

B. *Background Subtraction, Superwindowing and Resampling*

The background subtraction, superwindowing and resampling kernel (hereafter simply "resampling kernel") takes a batch of scans as an input. The block size of this kernel is varied, and, as stated in the previous section, it is found that for sufficiently large batch sizes there is little difference in performance. Figure 7 shows computation time for a single batch of scans as a function of block sizes. It is clear that for blocks larger than 2 threads, the performance is approximately constant. We choose a block size of 1,024.

As discussed in the previous section, the size of the input batch is similarly inconsequential past some minimum value. We test the runtime of this algorithm in serial on the CPU as compared to our kernel for varying input batch sizes, and the results are shown in Figure 8. We see that for suffieciently large batches, GPU performance is relatively constant while CPU runtime increases significantly. The batch will always be at least oone scan (2,048) point long, so the GPU will always outperform the CPU for this task. Thus, to determine optimal batch size, we must look at the other operations in the pipeline.

C. *Fourier Transform*

Although there is no way to define the block size, grid size or optimize the memory transfer of our FFT using Cupy, we still test the speed of the GPU and CPU FFT algorithms. Both perform row-wise FFTs, each row having 2,048 points, as the batch size increases from 21 to 216. The resulting runtimes are shown in Figure 9. Like in resampling, when the batch is small the CPU outperforms the GPU. However, as batch size increases, the CPU time starts to explode at about 2^9 while the GPU fft time starts to explode at 212, in our project, the batch size is 212, at which the CPU begins to slow down while the GPU does not.

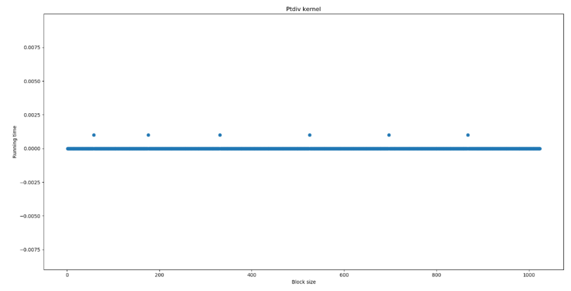

Fig. 5. Pointwise division kernel runtime by block size. The block size does not clearly affect the run time, except for some arbitrary aberrations.

D. *Spectral Domain Phase Microscopy*

Finally we consider SDPM, which is a simple pointwise operation kernel. Thus, we expect that the block size will not affect the speed beyond a certain minimum . Figure 10 shows the runtime of the SDPM kernel with varying block size $N \times N$. For two dimensional square blocks, the maximum size is 32×32. After a certain point, as expected, the performance does not change, so we use 32×32 blocks.

We also vary batch size and evaluate the running time of CPU SDPM and GPU SDPM algorithms. Figure 11 shows these runtimes as a function of batch size. The two curves begin to diverge at 2^9. For batch size 212, the CPU time is 0.046 seconds while the GPU time is about 0.0001 seconds, yielding a speedup of 460×. Thus SDPM does not have a large effect on total time taken in this processing pipeline.

E. *Total Timing*

The total timing of our parallelized SDPM processing, with the optimal parameters chosen above, is about one minute for a 5-second *in vivo* recording. The serial processing time for this same file is 20 minutes, suggesting a 20× speedup. The data loading and background fitting take the same time in either case, so time is saved almost entirely in the resampling and SDPM steps. The resampling step, as can be seen in Figure 8, offers a speed-up of approximately 20× at the batch size used. SDPM on the other hand takes negligible time compared to its serial counterpart – the GPU algorithm is almost $500\times$ faster.

VI. Conclusions

We have shown that parallel processing can be used to significantly accelerate the processing of OCT data for cochlear imaging *in vivo*. We have also found the parameters under which this accelerated algorithm performs best (that is, batch sizes, block sizes, etc.).

The use of batched parallel computations has achieved a huge speedup – about 20 times – over the currently used MATLAB algorithm. For long experiments, in which many data sets of this type are to be processed, it currently takes about one day to process all of the data with MATLAB. With our parallelized algorithm, this will only take about one hour,

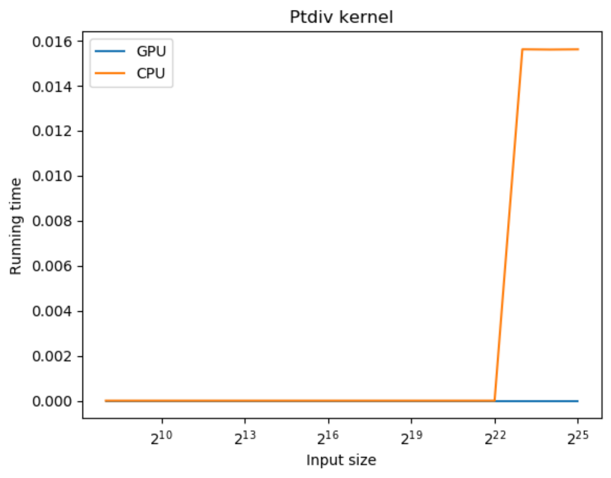

Fig. 6. Pointwise division runtime for CPU and GPU operations by input vector size. CPU and GPU times are nearly the same until vectors get very large – a scenario that will never occur in our problem.

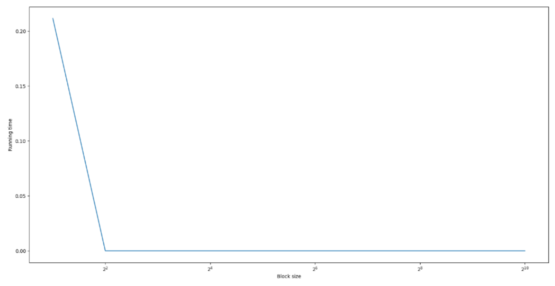

Fig. 7. Resampling kernel runtime by block size. The block size does not clearly affect the run time after a certain point.

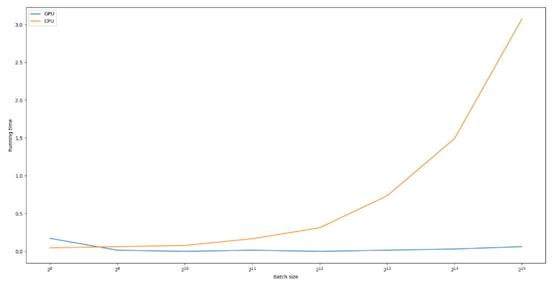

Fig. 8. Resampling runtime for CPU and GPU operations by input vector size. The GPU outperforms the CPU significantly at batch sizes larger than 2^9.

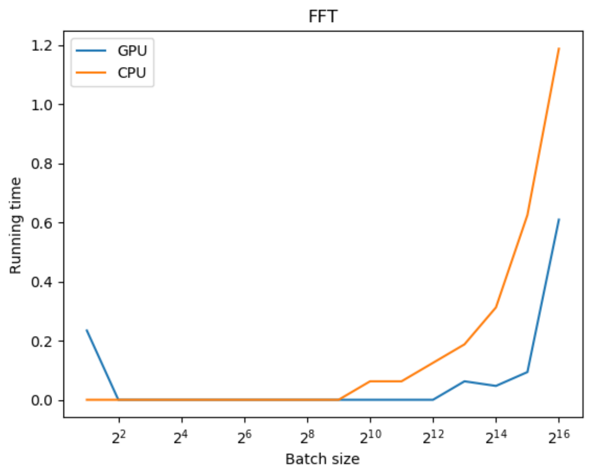

Fig. 9. Time taken by FFT kernel and numpy FFT algorithm with varying input size.

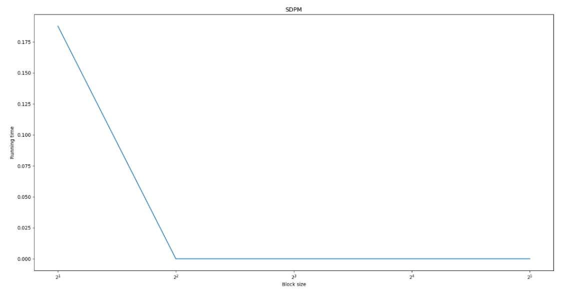

Fig. 10. SDPM kernel runtime by block size. The block size does not clearly affect the run time for sufficiently large block size.

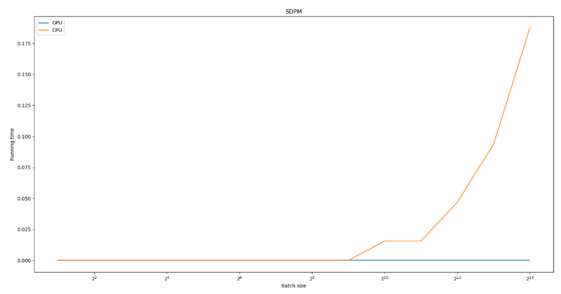

Fig. 11. SDPM runtime for CPU and GPU operations by batch size. The GPU outperforms the CPU significantly at batch sizes larger than 2^9.

greatly widening the bottleneck on our experimentation time incurred by processing.

About half of the time taken in this processing pipeline is through loading data into RAM from the RAW files. If this were implemented in the experimental pipeline rather than in post-processing, we could achieve a speedup of near 40 times. This would be, however, at the cost of about 30 seconds spent in the processing of each file during experimentation. Depending on the experiment, this could actually completely eliminate the need for post-processing entirely at little cost.

We have also shown that as the data sizes scale up – for example, if we were to use longer recordings within experiments – the GPU algorithm outperforms the serial algorithm by an even larger factor. This attests to the scalability of our kernels, and could open the door for longer-time (and thereby higher-SNR) experiments at the Fowler Lab in the future.

References

[1] E. Hecht, *Optics*. Pearson, 2017.

[2] W. E. Drexler, *Optical Coherence Tomography: Technology and Applications*. Springer International Publishing AG, 2015.