Improved Performance of CaFE and IRIS Model Fitting Using CUDA

Boston University College of Engineering
EC913: MS Project Report
Fall 2012

Dylan Jackson

ABSTRACT

Label-free optical bionsensors are known to be accurate and reliable tools for measuring and monitoring certain biomolecular interactions. In recent years, new techniques and technologies have emerged that enable high-throughput biosensing at lower system size, cost, and complexity. In particular, the LED-based Interferometric Reflectance Imaging Sensor (IRIS) has been demonstrated as a viable alternative to previously established high-end biosensors and its effectiveness can be augmented by the calibrated fluorescence enhancement (CaFE) technique. While IRIS enables high-throughput detection, its movement towards low-cost and field portability has introduced new processing challenges. Since the nature of the processing lends itself well to data level parallelism, it is natural to explore single instruction, multiple data (SIMD) architectures to improve processing performance. NVIDIA's CUDA technology enables parallel computing on compatible graphics processing units (GPUs) and is an ideal platform for exploiting data level parallelization. A CUDA implementation of the CaFE/IRIS model has been developed, yielding significant performance improvements over the current MATLAB implementation.

Table of Contents

1. Introduction 3
2. Increased Computational Complexity 5
3. Exploiting Data Level Parallelism 8
4. CUDA Implementation 13
   4.a. Overview 13
   4.b. Memory Spaces 16
   4.c. Calculation Method 18
5. Accuracy and Performance 21
6. Conclusion 25
7. Acknowledgements 26
8. References 27

1. Introduction

Optical biosensors are employed for a number of applications in a broad spectrum of fields. Pathogen detection and identification is crucial for health clinics and in-theatre military outposts alike, but traditionally has required time consuming and cumbersome laboratory techniques, specialized equipment, and highly skilled technicians [1]. Traditional laboratory methods involve testing and analyzing large sample volumes with techniques including complement fixation, polymerase chain reactions, and virus isolation by cell culture and immunocytological confirmation, all of which require laborious and precise sample preparation and processing [1]. While ultimately effective, the cost in both time and resources needed to deploy the proper facilities and perform the analysis makes the standard approach ill suited for extreme situations such as biological attacks or disaster scenarios which require rapid, high throughput response. Those same obstacles restrict the deployment of such diagnostic capabilities to high-end, well-established facilities with significant available infrastructure, making field analysis extremely impractical or outright impossible. Lastly, another disadvantage of traditional laboratory methods is their inherently low throughput.

The introduction of the microarray format for monitoring biomolecular interactions has improved upon some of the aforementioned obstacles, due to a "built-in" parallel detection capability [1]. Detection modalities have also progressed with the advent of microarrays. Biomolecular interactions are usually detected by using labels such as fluorophores, radioisotopes, and enzymes, but several problems with using

labels (due to cost, storage, uniformity and other issues associated with the labels themselves) have motivated the research and development of label-free detection methods [1]. Surface plasmon resonance (SPR) has emerged as a standard for optical label-free detection with comparatively high throughput capabilities. While an improvement upon the labeled methods, SPR still has drawbacks associated with it including high cost, low portability, and low multiplexing capacity, making it still impractical for rapid deployment and field use [1,2].

Another group of optical technologies have been recently exploited for monitoring molecular interactions by using various types of reflectance spectroscopy. Spectroscopic ellipsometry, Arrayed Imaging Reflectometry (AIR), Molecular Interferometric Imaging (MI2), and Reflectometric Interference Spectroscopy are all examples of platforms that have been used for this purpose and have been discussed in other works. An alternative to these approaches, the Interferometric Reflectance Imaging System (IRIS) offers comparable sensitivity to the SPR technique [2]. Using a silicon substrate with an oxide layer of known thickness, common path interferometry is used to monitor changes in the local optical path length. These changes (compared to a bare Si region) can be attributed to the accumulation of mass on the oxide surface [2]. Data is recorded using a CCD camera at multiple illumination wavelengths, and the pixels are iteratively fit to a wavelength dependent reflectance model using the Levenberg-Marquardt fitting algorithm. Originally this system utilized a tunable laser to illuminate the microarray with the desired wavelengths in order to measure the spectral nature of the reflectance, but the high cost and fragility of tunable lasers limit their

utility for low-cost or field-deployable systems. IRIS was further developed to use four LEDs instead of a tunable laser to illuminate the microarray. While this reduced system cost, it introduced additional complexities into the reflectance model.

2. Increased Computational Complexity

Using IRIS with a tunable laser, the pixels are fit against the reflectance model

$$R = \left| \frac{r_1 + r_2 \mathrm{e}^{-\mathrm{j}2k_z d}}{1 + r_1 r_2 \mathrm{e}^{-\mathrm{j}2k_z d}} \right|^2$$

where the reflectivity values r_1 and r_2, and wavenumber k_z are functions of the illumination wavelength, and d represents the optical thickness of the pixel. Using LEDs instead of a tunable laser, the model remains the same, but because LEDs have a much broader spectral linewidth a single wavelength can no longer be used to evaluate r_1, r_2, and k_z. Instead, it is necessary to integrate the reflectance over the spectrum of each LED. From an analytical standpoint this is a near-trivial change, but it increases the computational complexity significantly. Instead of calculating the model once for each wavelength, the model must be calculated for each sampling period of the LED's spectral linewidth, multiplied by a weight (corresponding to the normalized LED intensity at that wavelength), and then summed (integrated), giving the new equation the form:

$$R = \sum_{n=0}^{n_\lambda - 1} \beta_n \left| \frac{r_{1,n} + r_{2,n} \mathrm{e}^{-\mathrm{j}2k_{z,n}d}}{1 + r_{1,n} r_{2,n} \mathrm{e}^{-\mathrm{j}2k_{z,n}d}} \right|^2$$

Where β_n is the spectral weighting function and n_λ is the number of samples of the LED's spectral linewidth. The above as well as subsequent equations refers to a single LED, and thus must be calculated for each LED separately. The increase in computational complexity is directly proportional to the number of discrete steps within the LED linewidth: over 330 in the current instantiation of the model. Currently, this data must be processed on a large computing grid, as a single computer takes a prohibitively long time to perform the fit. While the device itself has become more suitable for low-cost portable implementations, the processing necessary to compensate for the design change still greatly limits the utility of IRIS outside of a high-end laboratory setting.

The computational load is compounded when the LED-IRIS system is augmented by the calibrated fluorescence enhancement (CaFE) technique. Measuring fluorescence is a powerful, well-established method of detecting labeled molecules and is performed by illuminating a sample using a high intensity source with a very narrow bandwidth, usually a laser. The labels – often fluorophores – emit light, which is captured by high numerical aperture (NA) optics – a necessity for adequate sensitivity. The CaFE technique uses a Si/SiO2 layered system, like IRIS, which significantly improves the signal return compared to the glass slides usually used [3]. While label free techniques such as LED-IRIS allow an accurate measurement of the amount of molecular probes

deposited on a microarray, fluorescence can determine how many labeled secondary molecules bind to the probes. A standard operational cycle would begin using LED-IRIS to measure the amount of probe material originally deposited on the microarray, then labeled secondary material would be added, and finally a separate device would illuminate the microarray to measure binding by fluorescence. In the spirit of moving towards portable and field-deployable systems, a system combining LED-IRIS with the capability to stimulate and measure fluorescence has been prototyped.

Although separate optical paths exist for the LED-IRIS and CaFE illumination sources, they must share a primary aperture to achieve space savings. The standalone LED-IRIS device used low NA optics at enough of a standoff such that all of the returning light effectively entered the primary lens at one incident angle. However, fluorescence requires high NA optics to capture as much light as possible, therefore a larger lens must be placed closer to the microarray for the measurement. Now the return signal to LED-IRIS can no longer be approximated as normal to the lens, and an angular weighting must be applied in addition to the spectral weighting. This adds yet another summation (integration) dimension to the LED-IRIS model. Another consequence of the angle dependence is with regard to polarization. At normal incidence to the objective, both reflected polarizations of light (s and p) have equal magnitude. With a high NA objective, off axis reflection is also collected and the two polarizations do not have equal magnitude at those angles. This condition not only adds another dimension of weighted summing (integration), but it also requires the model to be calculated for both s and p polarizations and averaged together to obtain a final result, as shown below

$$R_p = \sum_{m=0}^{m_\theta - 1} \sum_{n=0}^{n_\lambda - 1} \beta_{(n,m)} \left| \frac{r_{p1,(n,m)} + r_{p2,(n,m)} \mathrm{e}^{-\mathrm{j}2k_{z,(n,m)}d}}{1 + r_{p1,(n,m)} r_{p2,(n,m)} \mathrm{e}^{-\mathrm{j}2k_{z,(n,m)}d}} \right|^2$$

$$R_s = \sum_{m=0}^{m_\theta - 1} \sum_{n=0}^{n_\lambda - 1} \beta_{(n,m)} \left| \frac{r_{s1,(n,m)} + r_{s2,(n,m)} \mathrm{e}^{-\mathrm{j}2k_{z,(n,m)}d}}{1 + r_{s1,(n,m)} r_{s2,(n,m)} \mathrm{e}^{-\mathrm{j}2k_{z,(n,m)}d}} \right|^2$$

$$R = \frac{R_s + R_p}{2}$$

where m_θ is the number of angle (theta) steps. With 100 steps of theta, this now means the model must be calculated 200x more per fitting iteration. Even with an available grid, the CaFE variant of the IRIS model takes many hours or possibly days to compute; the true time is unknown because the computation takes so long that it has never been permitted to run through a full data set. Aside from the obvious drawback of requiring a grid, what began as an inconvenient analysis wait-time for LED-IRIS becomes an unacceptable processing latency for the combined LED-IRIS/CaFE setup.

3. Exploiting Data Level Parallelism

We have seen how easy it is for the computational complexity of a model calculation to grow by several orders of magnitude, despite relatively tame analytical changes. That fitting the data is an iterative process – meaning the model must be evaluated many times per pixel – is not of much comfort either. The Levenberg-

Marquardt algorithm seeks to minimize the sum-squared of errors (SSE) in the following (high level) steps [4]:

1. Evaluate the model
2. Calculate the SSE
    a. Change a "fudge factor" depending upon whether SSE increases or decreases, affecting the subsequent adjustment of input parameters
3. Evaluate the Jacobian of the function with respect to SSE (involving more model calculations)
4. Calculate a linear least-squares fit based on the Jacobian
5. Choose new input parameters based on the least-squares fit
6. Repeat until convergence criteria met
    a. (MATLAB default: change in SSE is < SSE*$1e^{-8}$)

The fit also adjusts an amplitude and offset factor as well, so the full model takes the form of a linear equation,

$$y = mR + b$$

where *R* is the reflectance as calculated in the previous section and *m* and *b* are the amplitude and offset values respectively.

Despite the lengthy iterative process, the same calculations must occur for each pixel using the same input values with the exception of the fitting parameter, *d,* which will change to fit each pixel. We can take advantage of this property using a single

instruction, multiple data (SIMD) computing architecture. SIMD refers to a type of computer that executes a single task or operation on a multitude of different data elements. While a large computing grid already acts as a SIMD system, it is possible to improve upon its performance by utilizing a more specialized processing platform such as a graphics-processing unit (GPU).

GPUs were originally developed to serve the needs of the video and computer game industry. In time this naturally extended to more general image processing as well as general-purpose parallel computation. A traditional central processing unit (CPU) is made up of primarily control logic and only a few *sequential* arithmetic logic units (ALUs). CPUs are optimized for low latency and flexible use, making them ideal for general computing but less optimal for high throughput calculations. GPUs are made up of hundreds of *parallel* ALUs, making them ideal for highly parallel computation (Figure 1).

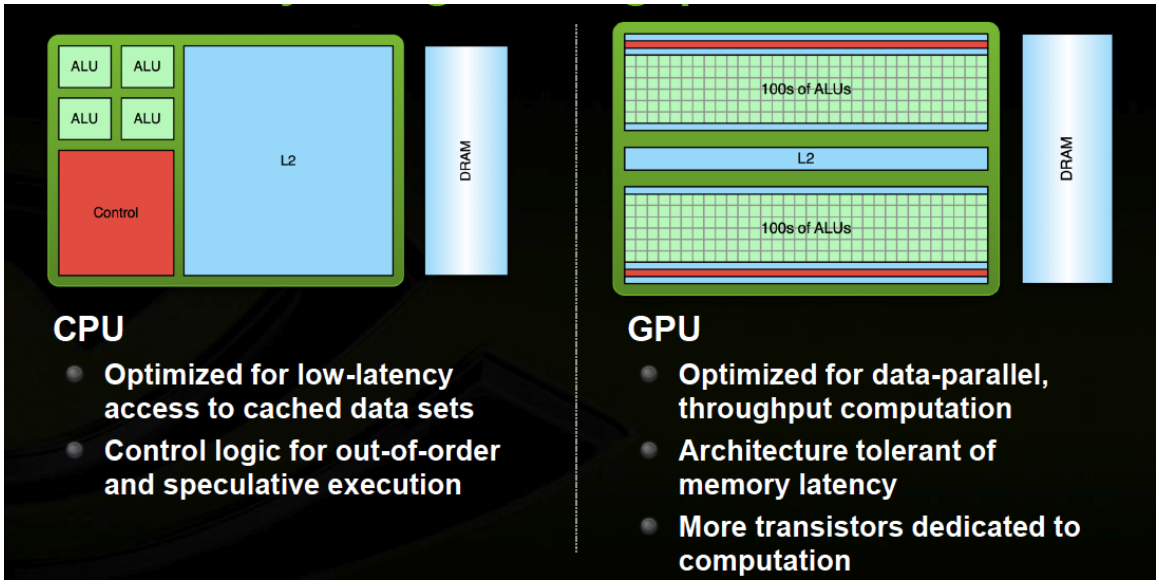

Figure 1 - CPU architecture vs. GPU architecture (Source: NVIDIA)

While it is not immediately obvious that high latency memory would be acceptable in a high performance computing, the GPU effectively hides the memory latency by computing on some threads while other threads wait for requested data (Figure 2). Of course, the GPU must have enough parallel work to do so that it *can* hide the latency. If a GPU were programmed to perform only serial operations, it would almost certainly perform much slower than a CPU due to the high memory latency.

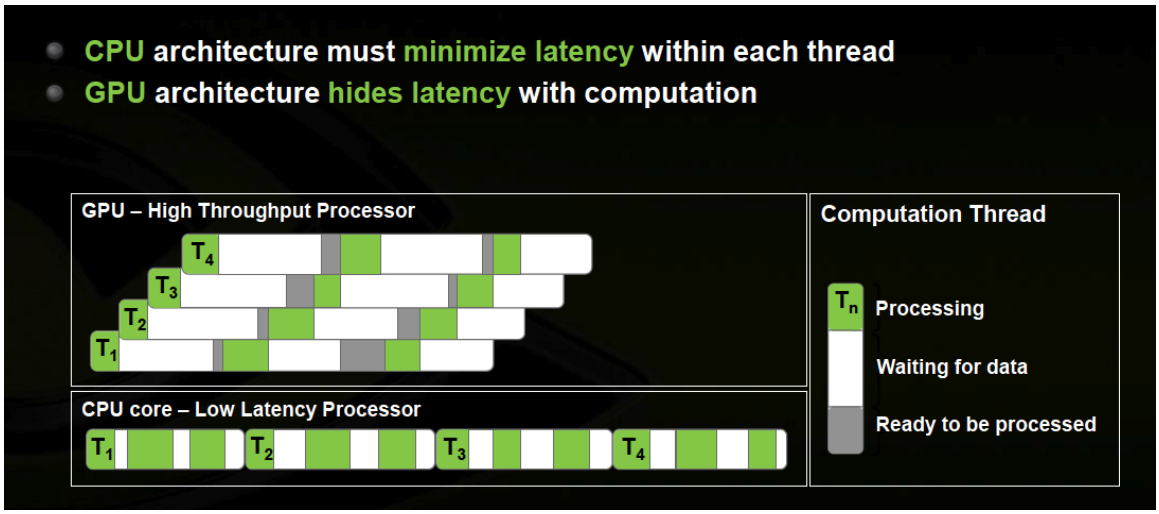

Figure 2 – GPU architecture processing flow (Source: NVIDIA)

Given the data-parallel problem at hand, it seems reasonable that a GPU would be an ideal processor for fitting the LED-IRIS/CaFE data to the reflectance model. In 2006, NVIDIA introduced a parallel computing platform and programming model to interface with compatible NVIDIA GPUs called CUDA. CUDA has C and C++ APIs, making it relatively straightforward to apply with knowledge of either language. CUDA's programming model also is, as would be expected, well matched to the data-parallel problem.

With CUDA, the functions that actually run on the GPU are called kernels, and are executed by thousands of threads simultaneously within the GPU's streaming multiprocessors (SMs). The threads are organized in "blocks" that execute independently from one another. Thread blocks are organized in a "grid" (Figure 3). When a kernel is launched, block dimensions (up to 3 dimensions) as well as grid dimensions (up to 2 or 3 dimensions, depending on the architecture) are specified in the execution configuration. All threads/blocks execute the same kernel, though they can be instructed to take different paths within the kernel. The blocks are initially distributed amongst all of the GPUs SMs, each of which can execute multiple blocks simultaneously; the remaining blocks are placed into a global queue. As blocks complete they leave their respective SM and new blocks from the queue take their places.

Figure 3 - CUDA Kernel Execution (Source: NVIDIA)

Regardless of block size or dimension, threads are grouped into "warps" of 32 threads and execute instructions in lockstep with each other. While at a high level all of the threads are considered to be executing in parallel, warps receive instructions (mostly) sequentially with respect to one another. A full warp carries out every instruction handed to it regardless of how many threads in that warp actually have work

to do. Typically it would be reasonable to execute an image-processing kernel such that each pixel has one thread assigned to it. In cases where each thread processes a pixel, this is usually fine. However a fitting algorithm executes on each pixel for an unknown number of iterations since some pixels will converge to a fit sooner than others. If a thread is assigned to each pixel, skipping finished pixels would have little effect on processing time; any warp with a single unfinished pixel would execute in the same amount of time as a warp full of unfinished pixels. This can be addressed by assigning one block to each pixel instead of one thread. If a block is skipped none of the warps in that block will do any work and another block can take its place in the SM, thus enabling the GPU to spend time processing only the pixels that have not yet converged to a fit. Because there are so many calculations necessary for each pixel, there is plenty of work available to keep all threads in a block busy. This is only necessary for kernels that invoke the model calculation. Although the same thread "wasting" occurs for finished pixels elsewhere in the fitting algorithm, there is no additional parallelism to be exploited by restructuring the execution; assigning a block to each pixel would be even more wasteful as an entire warp would be wasted on single-threaded serial operations.

4. CUDA Implementation

4.a. Overview

The work presented here is not the first time CUDA has been used for IRIS. Previous work included a C++ CUDA implementation of the Levenberg-Marquardt

algorithm, but it had a number of limitations. First, the original code only supported the original model that used discrete wavelengths. The original model calculation could be executed with only one thread per pixel, but the new models required over 300 times and over 60,000 times the number of calculations for LED-IRIS and CaFE respectively, necessitating new calculation code in addition to the different execution configuration. Secondly, the existing code did not support complex numbers. In the original model only magnitude was considered for the discrete wavelengths, but with the additional dimensions introduced by the LED implementation of IRIS and the CaFE technique the imaginary components also contribute to the result. Furthermore, use of different substrate materials in the future could give the reflected wavenumber an imaginary component.

Additionally, the original model was implemented with double precision floats. While not a drawback in itself, using double precision significantly reduces computation speed (the GFLOPs specification for double precision is often half that of single precision for CUDA GPUs) and increases memory and bandwidth usage. Although it enables higher precision calculations (as implied by the name), it is generally better to use single precision unless double precision is explicitly required. While the documentation that accompanied the original code asserted that double precision was required to meet the accuracy requirements, it is unknown what these requirements were and why single precision did not satisfy them. Because the fit converges to parameters based on the SSE of the output, minimization still yields only a statistical estimate [4]. Given the same statistical convergence criteria, neither double nor single precision is guaranteed to be

more or less accurate than the other for any given pixel. Double precision *could* allow for more stringent convergence criteria to be set but single precision supports the previously specified 1e-8 convergence criteria, which is already orders of magnitude smaller than what is statistically significant or usually necessary in practice [4].

From a practical standpoint, using double precision also restricts hardware compatibility to more expensive NVIDIA cards such as the Quadro and Tesla series. While double precision is technically supported in all CUDA GPUs of compute capability (CC) 1.3 and higher, the GeForce cards (targeted toward the PC gaming market) do not have their double precision cores enabled. Since gaming computation does not require double precision, disabling those cores allows the single precision cores to be overclocked without thermal overloading and hence delivering faster single precision computation. The Quadro and Tesla cards leave the double precision capability intact, but are more expensive because of specialized driver development and support for the business and industry end users these cards are developed for. This is not to say that Quadro and Tesla cards could not deliver equal or better performance than GeForce cards in general, but it is preferable to maintain platform flexibility especially to accommodate for hardware budget and availability. With these factors in mind, the LED-IRIS/CaFE implementation was written using single precision floats, a new thread/block/grid configuration, and complex math support.

All development and initial testing was performed using Microsoft Visual Studio 2010 on Windows 7 64-bit with the CUDA 5.0 toolkit. The development GPU was an NVIDIA GeForce GTX680, a card built on NVIDIA's new Kepler architecture.

4.b. Memory Spaces

It is often not sufficient to write a CUDA kernel the same way one might write a C function for a CPU. This is because most standard CPU code deals with only one type of memory space (not including non-volatile storage): host memory or RAM. CUDA GPUs have multiple types of memory spaces: global, shared, constant, register, and local (a subset of global). Utilizing each of these spaces properly is key to achieving peak kernel performance. The following explanations will provide high-level descriptions of the different memory spaces and information about how to use them effectively. There are numerous whitepapers, programming guides, and other documents that go into great detail about these types of memory; this is meant to be a quick summary. The term "device" refers to the GPU, while the term "host" refers to the CPU.

Global memory (sometimes called device memory) refers to the main memory on the GPU. Because of its size and lack of use restrictions, global memory is the best place to store persistent variables and arrays as well as large data sets. Global memory can be allocated and accessed using CUDA API calls from the host, as well as read from and written to within kernels. The primary drawback of global memory is its high latency (hundreds of clock cycles), but this can often be hidden with computation as previously discussed.

Shared memory exists within the SMs of the GPU. It is very limited in size, but has extremely low latency and is ideal for storing intermediate computation results; it is often used like a manual cache. When shared memory is used, it is divided up between as many blocks on the SM as possible based on the amount allocated for a single block.

Shared memory is local to a block, which is to say that all threads within one block see the same shared memory space. Shared memory is divided into memory banks – 16 or 32 depending on the architecture. Care must be taken to ensure that threads within a warp do not access the same banks at once which results in serialized reads (bank conflicts) unless all threads access only a single bank (warp broadcast). Because shared memory is accessible by all threads in a block, the threads must be synchronized between writes and reads to avoid a race condition.

Constant memory is special subset of global memory that is limited in size (64 kB), but automatically cached to improve performance. It cannot be modified from within a kernel, but can have new values copied to it using CUDA API calls from the host. Constant memory is ideal for storing frequently used coefficients or constants that do not change during computation. Constant memory was not used for this implementation of the LED-IRIS/CaFE model.

Each SM also has a certain number of 32 bit registers available. These registers are the fastest memory space available to the device within a kernel and are typically used when non-array variables are declared within a kernel. Registers are local to a *thread*, so a variable declared in a kernel exists for every thread and thus uses as many registers as there are threads. Due to their low latency, registers are also useful for storing intermediate calculations if available.

Local memory has two primary uses: accommodating for register spillage and storing arrays declared within a kernel. When there are not enough registers to hold all of the declared variables, they are stored in local memory. Local memory is named so

for its locality to an individual thread, but it resides in global memory and thus has the same high latency drawback unless cached. The Fermi and Kepler architectures (CC 2.x and 3.x respectively) cache stores to local memory, making light register spillage acceptable to a point. On pre-Fermi cards, local memory usage is much more expensive due to the lack of caching.

4.c. Calculation Method

In order to keep the model general and useful for both LED-IRIS and the LED-IRIS/CaFE device, the CaFE version of the model was implemented. For LED-IRIS the s and p polarizations are equal, so simply setting the s and p components equal to each other will yield the correct result at the cost of roughly twice the necessary calculations. Despite this redundancy, high performance is still achieved for LED-IRIS data while maintaining compatibility with CaFE data.

In the full model calculation, the same equation must be evaluated for each angle step of each wavelength step for each LED. This can be envisioned as a 3D matrix composed of 2D layers of angle by wavelength, with one layer per LED; the model must be calculated for each element of this 3D matrix. The matrix of calculated values must then be summed and weighted along the angle and wavelength dimensions to yield one value for each LED center wavelength. Because the weights and summations do not have to be applied in any particular order with respect to each other, they can be done incrementally over the entire matrix.

Before calculations, all variables (except *d*, which is a fitting parameter) are pre-calculated with respect to angle, wavelength, and LED in MATLAB, and then written out in binary files. The files are read in by the CUDA application, which copies them to global memory in a 2D format, where each LED has its own "row" consisting of the angle and wavelength components (each 2D angle by wavelength layer is reshaped as one row). Data dimensions and element order are consistent across all variables after MATLAB preprocessing, and thus remain aligned when copied to the device.

The model is executed by a thread block that is 32 threads (one warp) wide and *nLambda* threads high, where *nLambda* is the number of center wavelengths. Each row of threads performs the calculations for one LED corresponding to 32 elements of the angle/wavelength layer (hence the *nLambda* thread block height) including the appropriate weighting, and then adds the result to a running sum before stepping to the next 32 elements. This combines the weighting and summation with the model calculation to avoid looping through the entire data set more than once. When the thread block reaches the end of the model data rows, all threads in the block write their values to shared memory and are then summed in a loop over the block rows. This process occurs for each pixel and calculates the model correctly, as compared to the MATLAB calculation.

Local memory usage severely limited performance during early stages of development. The CUDA C API includes a special cuFloatComplex data type, which is a structure of two floats representing the real and imaginary components. This type serves as an input to a number of complex math API functions including multiplication,

absolute value, and others. Initially the model had been implemented using nested complex math functions, but these functions allocate several floats within them for intermediate values and often call each other. Without registers to spare, all of these intermediate floats were allocated in local memory. With all threads creating so many additional floats the cache could not keep up with the local memory stores, forcing the values into global memory. This effectively serialized each calculation due to the high memory latency. The cache misses and subsequent local memory usage were verified in PTX code (pseudo-assembly CUDA code generated by the compiler) and the Nsight profiler.

This problem was eventually resolved by two significant implementation changes. The first change was to split up the large nested expression into separate sequential expressions, which used a combination of registers and shared memory to store intermediate results. While this drastically improved performance, there were significant additional gains to be made by eliminating the complex functions altogether. This required explicitly coding each instruction for all complex mathematical operations. Though tedious, this proved extremely effective and enabled the real-time performance achieved on the development GPU. For this application, real-time is defined as a processing time less than or equal to the frame collection period (currently 1 to 2 minutes). There were a number of other instances in the fitting code where functions had to be replaced with explicit operations. Once it had been verified that the CUDA model was calculating the correct values, some part of the fitting algorithm was producing NaNs after a number of iterations. This was eventually traced to a single

pow(x,y) function, which simply raised a value (*x*) to the power of 2 (*y*). After replacing that function with explicit self-multiplication (*x*x*), the issue was resolved and the fitting algorithm ran successfully.

5. Accuracy and Performance

Once the local memory issues had been addressed and the intermediate calculations were moved between shared memory and registers, there remained few options to increase performance by any significant amount. Profiling indicated that the time dominant kernels – the fit and Jacobian kernels took the longest because they involved calculating the model – ran at 100% and 75% occupancy respectively. Occupancy refers to how many threads compared to the maximum are able to run concurrently in an SM. While it is important to seek a high occupancy, there is a certain point at which performance will not improve even if occupancy increases. The Jacobian kernel occupancy was limited by the register use; this means there were not enough registers available on a given SM to completely fill it with active thread blocks. Forcing a register use limit in the compiler enabled the Jacobian kernel to achieve 100% occupancy, but performance actually dropped as a result. In this case, variables the compiler would have placed in registers were forced to local memory, causing performance to suffer due to the high latency associated with local memory.

Enabling the fast-math directive in the compiler greatly improved performance as well. Using fast-math causes the calculations to break with strict IEEE compliance in

order to make certain optimizations, essentially trading some accuracy for speed. In this case, using fast-math improved speed by nearly 30%, but the mean deviation from MATLAB double precision results was generally unaffected (Figure 4).

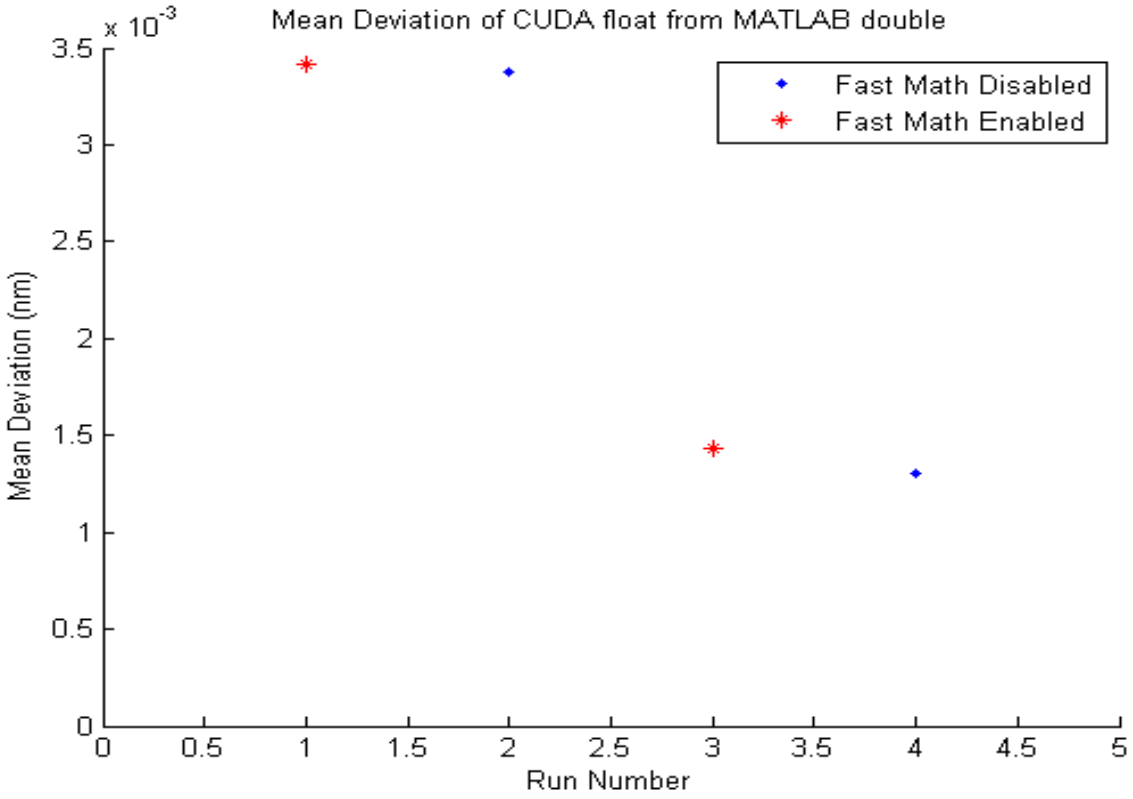

Figure 4 – CUDA result accuracy with respect to fast-math compiler directive. Runs 1 and 2 were performed with 1e-6 convergence criteria, runs 3 and 4 were performed with 1e-8 convergence criteria, and the deviation is with respect to MATLAB doubles with 1e-8 convergence criteria.

Considering the previously discussed statistical nature of the fit, a mean deviation differing by fractions of picometers is acceptable for the dramatic performance boost. Figure 4 also demonstrates that the results from the CUDA implementation deviate from the MATLAB results on the order of picometers. Given that the MATLAB version uses double precision and the CUDA version uses single

precision it is expected that there will be some small differences due to rounding, but both satisfy the same statistical fitting criteria.

Besides fast-math, no additional compiler directives were used beyond the default CUDA 5.0 settings in Visual Studio 2010. The following performance information was collected on LED-IRIS and CaFE data sets (1200 x 1600), using MATLAB's tic/toc functions for timing the release executable's runtime. Times are organized in Table 1 according to the GPU or CPU used, the model variant (LED-IRIS or CaFE), and the convergence criteria. Each GPU time is the average of 3 consecutive runs, rounded up to the next $\frac{1}{10}$ of a second. The CaFE calculation time with the GTX285 was not collected so as not to tie up a shared computer for an unknown period of time, but those can be estimated (indicated by *) based on the proportional differences between LED-IRIS and CaFE on the GTX680 and the LED-IRIS runtime on the GTX285. CPU grid time was estimated using an Intel i7-920 running MATLAB on 1/49th of a data set. Single CPU time is estimated by multiplying the estimated CPU grid time by 49 to represent a whole data set. All CPU-based runtimes were rounded up to the next second, or $\frac{1}{10}$ of a second if below 1000 s. It should be noted that the Boston University computing grid uses different processors, so the true grid runtime may be different than the estimate given below. Both CPU implementations use MATLAB's *nlinfit()* function.

Table 1 – Performance of IRIS/CaFE Fitting on Multiple Hardware Platforms

		Convergence Criteria		
		(1e-8) * SSE	(1e-6) * SSE	(1e-4) * SSE
Hardware Type	Model Type	Average Fit Runtime		
GTX 680 (CC 3.0) GPU	IRIS	44.8 s	39.5 s	22.1 s
	CaFE	4131.2 s	3193.6 s	1875.1 s
GTX285 (CC 1.3) GPU	IRIS	206.7 s	174.0 s	93.4 s
	CaFE	*19070 s	*14068 s	*7941 s
Grid of i7-920 CPUs (Estimated)	IRIS	525.9 s	322.2 s	308.9 s
	CaFE	16992 s	16525 s	14473 s
i7-920 @ 2.67GHz CPU (4 cores, 8 threads)	IRIS	25765 s	15785 s	15132 s
	CaFE	832608 s	809725 s	709177 s

The LED-IRIS and CaFE sets used were Chip1_URDataSet115006_0 and Chip1_URDataSet150635_0 respectively.

Compared to a single CPU fitting of LED-IRIS data (at the most stringent convergence criteria), the single GPU implementation improves performance by over 124x on the GTX285, and over 575x on the GTX680. Both cards also beat out the CPU grid by over 2x and 11x respectively. It is important to note the speed-up as the convergence criteria is reduced. Some applications may not require the 1e-8 criteria, and may benefit more from speed than accuracy, especially considering that SSE changes smaller than 1e-3 are usually not considered statistically significant [4]. The time scaling as it relates to convergence criteria will depend on the data (how many iterations each pixel requires), but it is clear that both GPUs outpace either of the CPU-based approaches. The GTX680 enables real-time processing for LED-IRIS at any of the tested criteria, considering images are often collected 1-2 minutes apart. The GTX285 reaches that territory at 1e-4, but still lags behind the GTX680 by 4-5x. The CaFE

calculation is still too slow to be used for real-time analysis on any of the tested platforms, though the CUDA implementation still yields a dramatic improvement in performance compared to the CPU-based approaches. Regarding the CaFE runtimes, it is apparent that the CaFE-to-LED-IRIS runtime ratio is much higher for the GTX680 than the CPU. According to NVIDIA's system monitor utility, the GTX680 is 99% utilized during both LED-IRIS and CaFE computation. However, the Windows Task Manager showed approximately 12-15% processor utilization during the LED-IRIS computation, compared to 75-85% processor utilization during CaFE. Coupling that information with the runtime ratios indicate that MATLAB appears to calculate the CaFE model more time-efficiently than the LED-IRIS model; this may be worth investigating in case a GPU cannot be utilized for any reason.

6. Conclusion

The continued development of the IRIS and CaFE devices has required revisions to be made to the original reflectance model. These changes have increased the number of necessary computations by orders of magnitude to compensate for the use of less expensive and more robust system components. Although originally requiring a large computing grid to fit the data, the use of a GPU enables faster processing on a single machine. The CUDA implementation of the general reflectance model – combined with an existing CUDA implementation of the Levenberg-Marquardt algorithm – outperforms the CPU cluster on both GPUs tested (the GTX285 and GTX680)

with the latter enabling real-time processing on LED-IRIS data. The tested GPUs span three generations of NVIDIA architecture (CC 1.3 to 3.0), indicating that the CUDA model calculation and fit will run on any GPU of CC 1.3 or higher as long as there is enough global memory available (over 512 MB are required for the current 1200 x 1600 images, at least 1GB is recommended). Given CUDA's history of maintaining cross-architecture compatibility, it is reasonable to expect the code to execute on future architectures without incident and likely with improved performance. Furthermore, the CUDA application will accept both IRIS and CaFE data of any size, angular/spectral resolution, or convergence precision (tested up to 1e-8) with processing time scaling accordingly. The use of CUDA has provided a long-term, scalable solution to the LED-IRIS processing problem and has significantly improved processing performance for CaFE. The ability to perform this processing on a single computer (local or remote) will make a field portable LED-IRIS or CaFE system more feasible as the technology is further developed.

7. Acknowledgements

I would like to thank Alexander Reddington and David Freedman for providing references and other information crucial to understanding the model, as well as providing low-latency, high throughput feedback during the course of this project. I also acknowledge NVIDIA for providing a GeForce GTX680 for testing and development.

8. References

[1] C.A. Lopez., G.G. Daaboul, R.S. Vedula, E. Özkumur, D.A. Bergstein, T.W. Geisbert, H.E. Fawcett, B.B. Goldberg, J.H. Connor, M.S. Ünlu, "Label-free, multiplexed virus detection using spectral reflectance imaging"

[2] G.G. Daaboul, R.S. Vedula, S. Ahn, C.A. Lopez, A. Reddington, E. Özkumur, M.S. Ünlu, "LED-based Interferometric Reflectance Imaging Sensor for quantitative dynamic monitoring of biomolecular interactions"

[3] M.R. Monroe, A.P. Reddington, A.D. Collins, C. LaBoda, M. Cretich, M. Chiari, F.F. Little, M.S. Ünlu, "Multiplexed Method to Calibrate and Quantitate Fluorescence Signal for Allergen-Specific IgE"

[4] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, "Numerical Recipes in C, Second Edition"

Though not specifically cited, the "CUDA Programming Guide" and the "CUDA C Best Practices Guide", both by NVIDIA, provided much of the information necessary to understand GPU functionality and to properly develop and apply CUDA to this problem.