View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CiteSeerX

Accelerating Phase Unwrapping and Affine Transformations for Optical Quadrature Microscopy using CUDA

Perhaad Mistry
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, U.S.A
pmistry@ece.neu.edu

Sherman Braganza
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, U.S.A
sbraganz@ece.neu.edu

David Kaeli
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, U.S.A
kaeli@ece.neu.edu

Miriam Leeser
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, U.S.A
mel@ece.neu.edu

ABSTRACT

Optical Quadrature Microscopy (OQM) is a process which uses phase data to capture information about the sample being studied. OQM is part of an imaging framework developed by the Optical Science Laboratory at Northeastern University. In one particular application of interest, the framework is used to extract phase information from the image of an embryo to determine embryo viability.

Phase Unwrapping is the process of reconstructing the real phase shift (propagation delay) of a sample from the measured "wrapped" representation which is between $-\pi$ and $+\pi$. Unwrapping can be done using the Minimum L^P Norm Phase Unwrap algorithm. Images are first preprocessed using an Affine Transform before they are unwrapped. Both of these steps are time consuming and would benefit greatly from parallelization and acceleration. Faster processing would lower many research barriers (in terms of throughput and performance) present when using OQM.

In this paper we report on accelerating Phase Unwrapping and Affine Transformations using NVIDIA's CUDA programming model. We also run elementary noise removal on the GPU using NVIDIA's CUBLAS (CUDA Basic Linear Algebra Subprograms) library. We integrate GPU execution into a Matlab environment to seamlessly interface to the pre-existing image acquisition system. By mapping the unwrap and noise removal to a GPU, and by also reducing the amount of I/O overhead, we are able to accelerate the end-to-end process by more than 7.3x. This enables our imaging framework to perform high speed image acquisition and visualization at near real-time rates.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming; J.3 [**Life and Medical Sciences**]: Biology and genetics

Keywords

GPGPU, Phase Unwrapping, Affine Transformation, CUDA, CUBLAS, Optical Quadrature Microscopy, Biomedical Imaging, Matlab External Interface

1. INTRODUCTION

Optical microscopy plays a key role in a number of important imaging applications. Optical Quadrature Microscopy (OQM) performs a phase unwrap to obtain the structure of objects being studied. Producing phase information in real time is a challenging problem; real-time phase unwrapping would allow researchers to visually inspect samples as they are being positioned on the microscope, and would greatly accelerate the rate of discovery in optical microscopy.

Many microscopy applications are limited by latency associated with viewing the specimen that is presently under the lens. For OQM, the CPU based version of the unwrap code takes more than 18 second to complete. Before unwrapping can be started, an Affine Transform is applied to the data, which adds an additional 3 seconds to the process. In order to improve the performance of both these data-parallel applications so that we can eliminate the latency in generating a visual display for the quadrature microscope, we map these applications to a GPU.

In prior work [15, 4], a range of phase unwrapping algorithms were studied. A number of C-based CPU implementations described in [4] were considered. These included the *Path-Following* and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Second Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU 2009), March 8, 2009, Washington DC, USA.
Copyright 2009 ACM ISBN 978-1-60558-517-8 ...$5.00.

Minimum Norm families of algorithms. The primary selection criterion for picking an algorithm is the quality of the unwrap for the data available. The Minimum Norm family of algorithms produced the smoothest solutions, but some variations like Preconditioned Conjugate Gradient (PCG) often experienced large errors [15]. The Minimum L^P Norm algorithm, which is also a member of the family of the Minimum Norm algorithms, produced the best overall solution (though at the expense of the greatest computation time). Since our criterion was to obtain the highest quality unwrap, we elected to use the Minimum L^P Norm algorithm.

It should be noted that the class of algorithms being considered for unwrapping are not unique to microscoscopy. These same techniques can be used in applications such as synthetic aperture radar interferometry and adaptive optics, although the focus of this paper is on medical imaging. All of these applications require the conversion of interferometry phase into propagation delay.

Previous work done on applying GPUs for 2D unwraps includes Karasev et al [8] who used GPUs to implement 2D phase unwrapping on NVIDIA GPUs using CG (C for Graphics), achieving a 35x speedup. They implemented a weighted least squares algorithm, similar to the PCG algorithm shown in Section 3.3.1, and applied it to Interferometric Synthetic Aperture Radar (IFSAR) data. Their implementation used multigrid and Gauss-Seidel iterations to solve the minimization problem. However, multigrid techniques do not work on our datasets [2]. Mutigrid techniques have been studied and they require a high number of iterations to converge (on the order of tens of thousands) [15]. In comparison, the PCG or Minimum L^P Norm algorithm used in this work requires tens or hundreds of iterations. Even though this prior work demonstrated impressive speedups, their total computation time is greater than the unwrapping times reported in this work.

The key contributions of this paper include:

- A description of a Minimum L^p Norm Phase Unwrap algorithm mapped to the NVIDIA GPU,
- A description of an Affine Transform map to the NVIDIA GPU,
- A performance evaluation of these two algorithms run on a CPU and a GPU and
- A demonstration of the utility of Matlab's MEX Interface to reduce the amount of disk activity.

The rest of the paper is organized as follows: In Section 2 we describe the microscopy system and the steps involved in Optical Quadrature Microscopy. We discuss the algorithms used for Phase Unwrapping and the Affine Transform and explain how they fit into the imaging framework in Section 3. In Section 4 we introduce how general purpose processing can be carried out on GPUs. In Sections 5 and 6 we describe how we implemented our algorithms on the GPU. We present performance results in Section 7, present future work in Section 8 and conclude the paper in Section 9.

2. OPTICAL QUADRATURE MICROSCOPY

In this section we describe the steps involved in OQM and also provide some background about the biomedical application targeted for our GPU-accelerated phase unwrapping.

2.1 Usage of OQM

The main motivation for improving the performance of OQM lies within its potential application in In Vitro Fertilization (IVF). Present imaging techniques used in IVF clinics are unable to produce accurate cell counts in developing embryos past the eight cell stage. These cell counts are necessary in order to study the viability of the embryo, as part of a program to understand fertility. The Optical Science Lab at Northeastern University has developed a method that has produced accurate cell counts in live mouse embryos ranging from 8 to 26 cells by combining two microscopy techniques: 1) Differential Interference Contrast (DIC) and 2) Optical Quadrature Microscopy (OQM).

The Optical Quadrature Microscope (OQM) technique was recently patented [6]. OQM can image unstained transparent objects, such as mammalian embryos, using very low, non-toxic, light levels. The contrast is in the index of refraction, which is different for culture media, cells, and intracellular components. OQM is an interferometric imaging modality that measures the amplitude and phase of the signal beam that travels through the embryo. The phase is transformed into an image of the optical path length difference, which is used to determine the area of maximum optical path length difference in a single cell.

The Optical Science Lab has developed an algorithm [16] to count the number of cells in late-stage pre-implantation embryos. The algorithm uses DIC microscopy for obtaining cell boundaries. The algorithm basically fits an ellipse to the boundary of a single cell using the DIC image and combines it with the optical path length deviation of a single cell that is obtained from the OQM image. This creates an ellipsoidal model of the optical path length deviation which is studied to produce the cell count and subsequently used to study IVF. Discontinuities are counted since they denote the presence of a cell or a change of the medium. DIC is another microscopy technique which does not require much post-processing. So DIC is not discussed further in this paper and our focus is on accelerating OQM.

2.2 OQM Setup

The OQM method is used in conjunction with other techniques [16] for imaging live embryos. OQM uses a 632.8 nm laser within a modified Mach-Zender Interferometer. Figure 1 presents the main elements of the OQM setup used to acquire images of a sample based on the optical path difference induced by the sample.

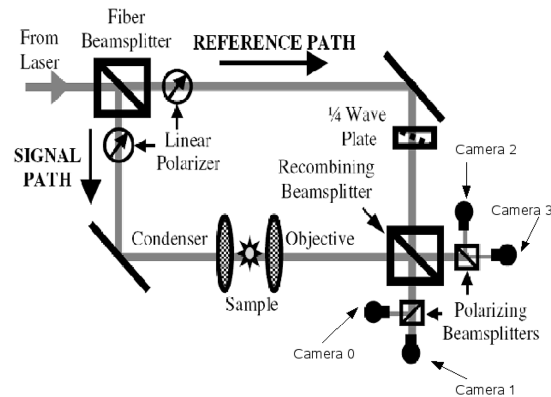

Figure 1: Organization of the Optical Quadrature Microscope.

As shown in Figure 1, the laser beam is split into two components called *reference* and *signal*. The signal beam passes through the sample. The reference beam is split, with one component being phase shifted by 90 degrees. The signal beam is then mixed separately with both components of the known reference signal. The merged signal that consists of the signal beam and non-phase shifted reference is called the *I channel* (the *in-phase signal*), The unknown signal that is mixed with the 90 degree phase-shifted reference signal is called the *Q channel* (the *quadrature signal*. By interpreting the I and Q signals as real and imaginary values of a complex number, we can find the amplitude and phase of the unknown signal. Expressions for both the reference and the signal beam are given by Equations 1 and 2.

$$E_{ref} = E * e^{(jwt+\phi)}(x + jy) \tag{1}$$

$$E_{sig} = E * e^{(jwt+\phi+\alpha)}(x + y) \tag{2}$$

When acquiring images, we record images from four CCD cameras simultaneously. The beam-splitters and the respective CCD cameras in Figure 1 capture the following fields, which are a combination of Equations 1 and 2.

$$\textit{Camera 0: } |E_{ref}|^2 + |E_{sig}|^2 + 2Re(E_{ref}.E_{sig}*) \tag{3}$$

$$\textit{Camera 1: } |E_{ref}|^2 + |E_{sig}|^2 + 2Im(E_{ref}.E_{sig}*) \tag{4}$$

$$\textit{Camera 2: } |E_{ref}|^2 + |E_{sig}|^2 - 2Re(E_{ref}.E_{sig}*) \tag{5}$$

$$\textit{Camera 3: } |E_{ref}|^2 + |E_{sig}|^2 - 2Im(E_{ref}.E_{sig}*) \tag{6}$$

As shown in Equations 3-6, and by inspecting the OQM diagram in Figure 1, we can see that by blocking the signal and reference arms individually and simultaneously, we are able to capture images for the pure signal (S_n, reference (R_n, and detector dark voltage (D_n. Equation 7 yields the complex number whose angle captures the phase of the sample [16].

$$E_r = \frac{1}{4} * \sum_{n=0}^{3} i^n . \frac{M_n - S_n - R_n}{\sqrt{R_n}} \tag{7}$$

The phase information produced from equation 7 produces wrapped phase-based images, with value at each pixel between $-\pi$ and $+\pi$. This data needs to be unwrapped in order to be usable, since we are interested in propagation delay not the wrapped/interferometry phase for the sample. Performing unwrapping with a C/Matlab implementation of the Minimum L^P Norm phase unwrapping algorithm takes nearly a minute to process a single frame. Accelerating this processing would render the OQM imaging modality much more useful in processing large stacks of images and provide near real-time performance to visualize unwrapped output from the microscope.

3. ALGORITHMS USED IN OQM

As described above, since we cannot directly measure the phase of an electrical field, we obtain the magnitude of different images from the cameras. We process the magnitude values to obtain the phase. The three main to obtain the optical path difference are:

1. An Affine Transform is run on the images obtained from the four frame grabbers for alignment purposes,
2. A noise removal step is performed, and
3. Phase unwrapping is performed to obtain the optical path difference from the phase difference.

3.1 Affine Transformation

Phase Unwrapping converts phase data into path difference information by looking at the phase difference between neighboring pixels. An Affine Transform [7] is used to compensate for imperfections in the acquisition equipment. Since data present at the same coordinates in images from two different cameras may not correspond to the same location in the real sample, so an Affine Transform is applied directly after images are collected. If we were to use images omitting this step, the phase gradients that we would compute in the unwrapping step would not correspond to neighboring pixels.

An Affine Transform involves operations including translation, scaling, rotation, skewing and reflection. As described [7], the transform can be expressed using matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{8}$$

Equation 8 shows that an affine transformation can be described as a matrix multiplication problem, where the location at which a data point is stored is decided by the result of a multiplication of the old coordinates of the point with a transformation matrix.

The new coordinates of each pixel (x',y') on the image can be expressed in terms of the matrix in Equation 8 and the old coordinates. We use the values of the 2x2 matrix obtained from the OQM frame grabbers. Only the final values of the 2x2 and 2x1 matrices are available.

3.2 Noise Removal

After the Affine Transform is applied and the images from the four cameras are aligned on a 2D grid, noise removal is performed. Noise removal is needed because of the presence of dark detector voltages and the fixed-pattern levels of the individual cameras which are caused by using non-ideal components. The dark detector voltage is removed by subtracting a dark image (image grabbed without the laser) from the sample image. The noise introduced by nonideal components is removed on a per camera basis and requires subtraction of S_n and R_n. The square root of R_n computed in Equation 10 normalizes each component to ensure balanced intensities for detection.

After computing Equation 10, we obtain images where the change in phase is only due to the sample being imaged. The images from each camera (the camera number denoted by n in Equation 10) are combined in Equation 10. This yields a complex number, and by calculating the angle, we extract the phase of the sample. This is the wrapped phase that is then passed to the phase unwrap code.

$$M_n \leftarrow M_n - D_n \tag{9}$$

$$E_r = \frac{1}{4} * \sum_{n=0}^{3} i^n . \frac{M_n - S_n - R_n}{\sqrt{R_n}} \tag{10}$$

3.3 Phase Unwrapping

In an ideal situation, the phase of an unwrapped signal varies such that the gradient between pixels is less than a half-cycle, or π radians. If this is true, then a wrapped version of this image may be unwrapped by simply summing (i.e. integrating in the continuous domain) until a gradient of π is reached at which point the phase is added to an integer multiple of 2π and the summation continues. This is the method for solving 1D phase-based data sets.

In 2 dimensions and if the data is noisy, phase gradients greater than π are created due to noise. These large phase gradients can lead to image corruption over large segments of the data. Even low levels of noise (i.e., below π) lead to an accumulation of error that eventually results in large deviations. Such errors lead to the presence of residues. A residue is defined as a point where the integral over a closed four pixel loop is not zero.

As mentioned earlier, we use the Minimum L^p Norm Algorithm for the unwrapping. This algorithm seeks to generate a solution whose local phase gradients match the original wrapped phase gradients as closely as possible. In order to describe the Minimum L^p Norm algorithm, we first neeed to discuss the conjugate gradient method, as well as the preconditioning that is applied to the image, since these two elements constitute the core computation of the Minimum L^p Norm algorithm.

3.3.1 Preconditioned Conjugate Gradient

The Preconditioned Conjugate Gradient algorithm generates iterations of the **unweighted** least squares algorithm in order to perform a **weighted** phase unwrap. The unweighted least squares algorithm minimizes the difference between the gradients of the wrapped phase data ϕ and the gradients of the unwrapped phase data Δ. Our goal is to obtain the $\phi_{i,j}$ that minimizes Equation 11.

$$\begin{aligned}\epsilon^2 = &\sum_{i=0}^{M-2}\sum_{j=0}^{N-2}(\phi_{i+1,j} - \phi_{i,j} - \Delta^x_{i,j})^2 \\ &+ \sum_{i=0}^{M-2}\sum_{j=0}^{N-2}(\phi_{i,j+1} - \phi_{i,j} - \Delta^y_{i,j})^2\end{aligned} \tag{11}$$

where $\phi_{i+1,j} - \phi_{i,j}$ is the phase difference in the x direction between the points $(i+1, j)$ and (i, j) of the unwrapped phase and $\Delta^x_{i,j}$ represents the phase difference in the x direction in the wrapped phase. The reduced expression can rewritten and discretized using the Poisson equation given by:

$$(\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}) + (\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}) = \rho_{i,j}, \tag{12}$$

where

$$\rho_{i,j} = (\Delta^x_{i,j} - \Delta^x_{i-1,j}) + (\Delta^y_{i,j} - \Delta^y_{i,j-1}) \tag{13}$$

Equation 12 solves the unweighted problem and is used to precondition the image matrix prior to running the conjugate gradient method. Preconditioning gives the matrix a better condition number allowing it to converge rapidly. Equation 12 is solved in the frequency domain using a 2D Discete Cosine Transform (DCT) and Inverse Discrete Cosine Transform.

The DCT algorithm selected uses a Discrete Fourier Transform (DFT) [9]. We utilize a GPU based FFT library called CUFFT provided by NVIDIA [12]. Pseudocode for the preconditioning step (which is part of a conjugate gradient method) is shown in Algorithm 1.

Algorithm 1 Preconditioned Conjugate Gradient

```
for k ← 0 To Max Iterations-1 do
  Apply Preconditioning
  Do Conjugate Gradient Steps
  Check For Convergence between Old and New Values
  if Convergence then
    Exit Loop
  end if
end for
```

3.3.2 Minimum L^p Norm Phase Unwrapping Algorithm

The Minimum L^p Norm algorithm minimizes the number of discontinuities in the unwrapped solution without concern for the magnitude of these discontinuities [4]. The Minimum L^p Norm algorithm can be used with or without user-supplied weights that denote the importance or quality of a pixel. The Minimum L^p Norm Algorithm iteratively runs the PCG algorithm, which in turn calls the DCT code.

The Minimum L^p Norm is similar to the PCG method since it also aims to minimize the difference between the gradients of the measured and calculated phases. The quality of an unwrap can be measured in terms of the number of residues in the image. A residue is defined as a point where the integral over a closed four pixel loop is not zero. PCG sets $p = 2$ or uses the least squares norm(Equation 11) whereas the Minimum L^p Norm algorithm uses $p = 0$ as seen in Equation 14. The value of $p = 0$ generally produces the best solution [4] This means that the Minimum L^p Norm algorithm minimizes the number of points where gradients of the wrapped phase differ from the unwrapped phase whereas the PCG algorithm minimizes the square of the differences. Equation 14 represents a weighted phase unwrap problem where the matrices U and V represent automatically generated data-dependent weights that indicate the quality of the phase data at particular pixel locations.

$$\begin{aligned}(\phi_{i+1,j} - \phi_{i,j})U_{i,j} + (\phi_{i,j+1} - \phi_{i,j})V_{i,j} - \\ (\phi_{i,j} - \phi_{i-1,j})U_{i-1,j} - (\phi_{i,j} - \phi_{i,j-1})U_{i,j-1} \\ = c(i,j)\end{aligned} \tag{14}$$

Here U and V are data-dependent weights and c is the weighted phase Laplacian given by

$$\begin{aligned}c(i,j) = \Delta^x_{i,j}U(i,j) - \Delta^x_{i-1,j}U(i-1,j) \\ +\Delta^y_{i,j}V(i,j) - \Delta^y_{i,j-1}V(i,j-1)\end{aligned} \tag{15}$$

This equation can be rewritten as a matrix equation, as shown in Equation 16.

$$Q\phi = c, \tag{16}$$

This equation takes the same form as the PCG equation, and so can be solved using the same methodology as presented in Section 3.3.1. The Algorithm for the Minimum L^P Norm is described

in Algorithm 2.

Algorithm 2 Minimum L^p Norm Phase Unwrapping Pseudocode

for $k \leftarrow 0$ *To Max Iterations* **do**
  Compute Residual R
  Exit it No Residues
  Compute data dependent weights U and V
  Compute weighted phase Laplacian c
  Call PCG (Algorithm 1)
end for
if No Residues Left **then**
  Unwrap using any simple algorithm like Goldstein
else
  Do Post Processing or Error Out
end if

4. CUDA INFORMATION

We provide an overview of NVIDIA's Compute Unified Device Architecture model (CUDA) [13] in order to justify the use of a GPU in our application. We use the NVIDIA 8800GTX GPU which supports the CUDA programming model. The microarchitecture of the system is as shown in Figure 2. The G80 GPU consists of a large number of streaming multiprocessors (SMs) - 16 for the 8800GTX version- which access a global memory. The GPU's *unified architecture* and the low-latency shared memory, make it possible to run non-graphics programs easily, thus harnessing the GPU's tremendous computing power for data parallel applications.

The typical model for utilizing a GPU is to treat it as a co-processor. The application developer then has to re-write his/her "kernel code" in order to execute on the GPU The host processor transfers input data to the GPU memory before executing the kernel and sends the output data back after the kernel completes execution. This programming model provides asynchronous concurrent kernel execution where control returns to the calling program as soon as the kernel is called. The hardware specifications of the NVIDIA 8800GTX used in this work are provided in Table 1.

	NVIDIA GeForce 8800GTX
# Multi Processors	16
# Scalar Processors	128
Shader Engine Clock	1500 MHz
Memory Clock	1080 MHz
Memory Interface	384 bits
Memory Bandwidth	103.7 GB/s
Peak Performance	320GFLOPS

Table 1: Hardware specifications of NVIDIA 8800 GTX.

The specifications of the host system is presented below in Table 2.

Programming Model: CUDA follows a data parallel programming model where each thread works with its own data. In order to specify the number of threads we must provide an *execution configuration* to CUDA. The execution configuration specifies the "grid" which consists of a number of threads divided into equally sized blocks. CUDA allows a programmer to index each thread in the grid via *thread id* and *block id*, which are accessible via keywords `threadIdx` and `blockIdx`.

CUDA provides two levels of data parallelism: 1) fine-grained data parallelism due to multiple threads within a block that access the low-latency local shared memory and execute independently, 2) coarser-grained parallelism (i.e., thread parallelism) that is available in the different blocks of threads that can be executed on different SM's and can access the larger global memory (though with more latency). The programmer must first partition the problem into subproblems which can be solved cooperatively through shared memory for maximum performance.

The CUDA programming model is commonly described as **SIMT** (single instruction multiple thread) [13]. Each thread is mapped to a single scalar processor core of the SM. Each thread is able to execute independently (i.e., with its own instruction address and register state) *Warps* are defined as a group of 32 threads that are created, managed, and scheduled by the SIMT unit with little overhead. To an application programmer, CUDA programming consists of extensions to ANSI C where the GPU is treated as a co-processor. The NVIDIA NVCC compiler seperates the code to run on the GPU or the CPU, though it is guided by simple C-like constructs provided by the programmer.

Component	Value
Processor	Core 2 Duo (Penryn)
L1 Data Cache	2x32kb
L2 Cache	6MB
Frequency	3Ghz
Number of Cores	2
RAM	4GB DDR2
Front Side Bus	4X 333M

Table 2: Host system configuration.

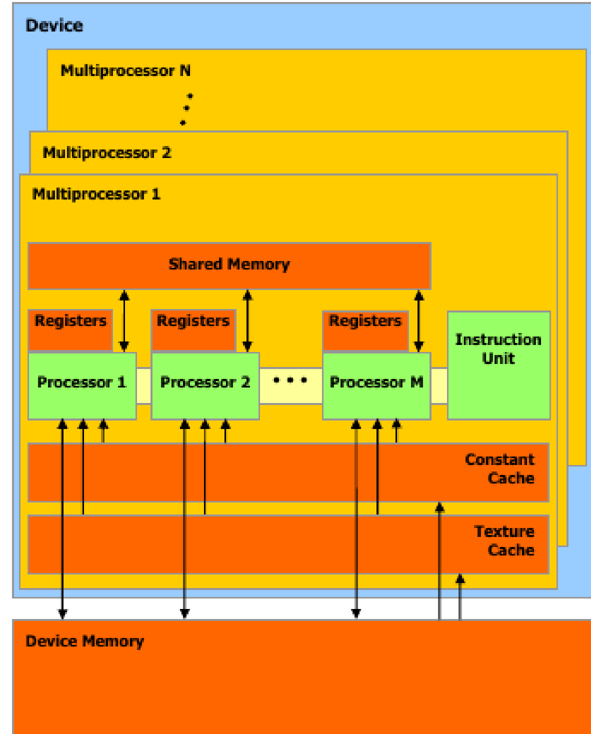

Figure 2: CUDA Hardware Model

We take advantage of these properties of the GPU in our application since the Affine Transforms and noise removal both are data parallel applications with the same operation is done on each pixel. For these applications we define a grid to be the size of the image and process different parts of the image in different parallel blocks on the multiple SMs on the GPU. Phase unwrapping also involves some element-wise operations including complex multiplication.

5. CUDA IMPLEMENTATION

In this section we explain how we implement the algorithms described in Section 3 in the OQM process on the GPU.

5.1 Affine Transformation

The Affine Transform described in Equation 8 is used to align the images obtained from the different cameras (see Figure 1). A naive implementation of an affine transform on a GPU is rather straightforward to code. As seen in the snippet shown in Algorithm 3, we compute one pixel using one thread. An important feature of this kernel lies in the method used to obtain the (x,y) coordinates for each pixel from each thread's block id and thread id. This removes the need for moving a precomputed coordinate system grid from the CPU to the GPU. Using the thread number $((BLOCKID)(BLOCKSIZE) + ThreadID)$ we get the (x,y) coordinates of a pixel which we multiply with the 2*2 Matrix to get the new coordinates.

Algorithm 3 Affine Transformation CUDA kernel pseudocode

Below Code Represents Contents of One Kernel
x ←*f(Blockidx.x, Threadidx.x)*
y ←*f(Blockidx.x, Threadidx.x)*
$PixelValue$ ←*Image(x,y)*
x' ←*ax+by+*α
y' ←*cx+dy+*β
As Per Equation 8
Image(x',y') ←PixelValue

Scatter & Gather Operations: In the psueudocode for Affine Transform (Algorithm 3) we see that the location of the result is decided by the value of the Affine calculations. This operation does not allow us to coalesce the memory accesses since the access patterns are not known apriori. This is a common data parallel access problem called "scatter". Scatter operations write data to arbitrary locations and "gather" operations read data from arbitrary locations. These operations are highly memory intensive. This problem has been studied in the past on GPUs [5]. We have observed this problem in our own application, but have not yet applied the methods shown in [5]. The method described [5] to deal with the memory bandwidth bottleneck implements a multi-pass methodology for scatters to improve data locality and increase coalescing possibilities. The benefits of a multi-pass technique [5] are noticeable for data sets that are greater than 10MB in size. A single matrix operated on by our kernel is only $640 * 480 * 4 = 1.2MB$. For this reason even while using a naive Affine Transform implementation, we achieve reasonable speedup since each thread performs a scatter for a single element, and this is only done once for each kernel execution.

5.2 Noise Removal

As discussed in Section 3.2, noise removal is done by using Equations 9 and 10. Writing kernels that perform point-wise subtraction would be trivial. However, due to the presence of very low-arithmetic intensity (i.e., the ratio of the amount of computation to memory accesses) in a kernel containing just a few subtractions, a naive implementation is not a good candidate since memory latency cannot be hidden easily. Instead we use the high performance library for CUDA called CUBLAS. This is an implementation of the Basic Linear Algebra Subprograms (BLAS) library that takes advantage of the data parallel hardware on the NVIDIA GPU. The performance difference between subtraction with BLAS and a naive kernel is shown in Figure 3, where we do a simple Saxpy operation $X \leftarrow X + \alpha * Y$ using the BLAS function $SAXPY(x, y, \alpha)$ and compare it with a naive kernel. We implemented the subtraction expressions of equation 10 using a set of SAXPY calls by setting $\alpha = -1$.

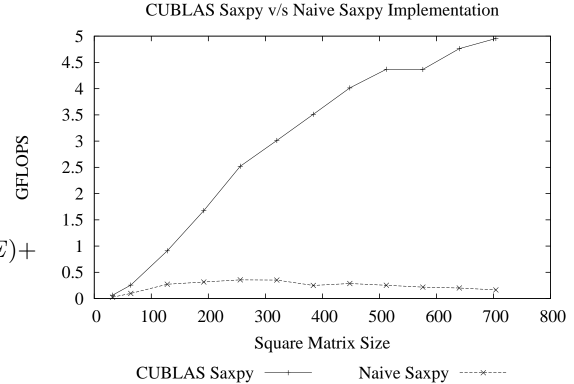

Figure 3: Comparing CUBLAS and naive implementations of SAXPY.

As seen in Figure 3, the usage of optimized BLAS is justified for noise removal, and the fact that data is already available on the GPU and the resultant data will be needed for phase unwrapping on the GPU, makes it worthwhile to use the GPU for an Affine Transform even for the unoptimized scatter algorithm.

5.3 Phase Unwrap GPU Implementation

Our GPU implementation uses a combination of NVIDIA supplied libraries and some custom kernels to implement the preconditioner and the conjugate gradient calculation, along with kernels specific to the Minimum L^P Norm calculation. Both the L^p Norm and PCG algorithm kernels were implemented on the GPU. This eliminates much of the data transfer between successive iterations as compared to implementing only the preconditioner.

As shown in Algorithm 4, it became necessary to also implement kernels that performed the two-dimensional shuffle and complex multiplication in order to convert between the FFT and the DCT. We see that the FFT calculation is the most time-consuming part of the DCT computation, followed by the shuffle.

5.3.1 Preconditioner Implementation

The preconditioner uses a 2D DCT/iDCT and some floating point computation in order to transform the input matrix into a form that converges rapidly when the conjugate gradient method is used to solve the equations presented in Section 3.3.1. The algorithm used for the DCT [9] focuses on the reuse of an existing 2D FFT. In our case, we use the CUFFT library provided by NVIDIA for implementing a DFT on a GPU [12]. CUFFT provides a complex Fourier transform that leverages the parallelism available on GPUs to rapidly compute 1D, 2D or 3D transforms. It is similar to the

Algorithm 4 The Minimum L^p Norm algorithm pseudocode.

```
for k ← 0 To Max Iterations do
  Calculate Derived Weights
  Calling PCG Method
  for j ← 0 To PCG Iteration Count do
    //Started Preconditioning Steps shown below
    DCT To DFT Steps (Shuffle Kernel)
    Execute CUFFT Call to do DCT
    Point-Wise Complex Multiplication to Get DCT Result
    Scaling Step
    Execute CUFFT Call to do iDCT
    //Finished Preconditioning Steps
    Conjugate Gradient Steps as in Algorithm 1
  end for
end for
if No Residues Remain then
  RasterUnwrap()
end if
```

popular FFTW library [3] since it uses a plan based approach to setting up and executing FFTs.

The first step of the 2D DCT is a two-dimensional shuffle. This shuffle reorders the input matrix in four different ways depending on the location of the individual data point. This procedure is not compute bound and is only limited by the performance of the memory bus and the efficiency of scatter/gather operations. The complex multiply step represents a straightforward kernel that is easily parallelized since each matrix value can be operated upon independently. This was implemented in CUDA using a thread-per-pixel model which assigns a thread to each matrix data point.

5.3.2 Point-Wise Multiplication

There are several point-wise matrix multiplications/additions and matrix accumulates. Point-wise functions parallelize extremely well since there are no dependencies between data points. To implement these functions, a similar method to the complex multiplication was used.

5.3.3 Reduction Kernels

An accumulation kernel was needed in order to compute the average magnitude of the elements. This was more complicated than expected since there are dependencies between elements inherent in accumulation and so it cannot be parallelized to the same degree. In stream processing terminology, this operation is called a reduction since the number of threads goes from N^2 for a N xN matrix to 1. This operation was frequently used and so it was necessary to optimize it further. A number of standard techniques [10, 14] were used including conflict-free sequential addressing, maximal thread utilization and to completely unroll loops.

6. SYSTEM INTEGRATION AND AUTOMATION

The OQM framework has a very heterogeneous structure in the sense that we have different systems that collaborate to solve a problem. Four Matrox frame grabbers are used for obtaining the images. Matlab and the Image Acquisition Toolbox are used to convert the image into a numerical matrix and provides a framework to view and save resultant images.The GPU code needed to be integrated into this environment.

Previously, in OQM the image acquisition process involved running the Affine Transform and also noise removal. These operations were done in Matlab. After the noise removal was complete, the data would be saved on the disk and the C-based implementation of the Phase Unwrap would be launched. Typically, the user would not use the saved data again. This would lead to unnecessary disk activity since data would need to be written by the Affine Transformation and noise removal steps and then again read for the Phase Unwrap.

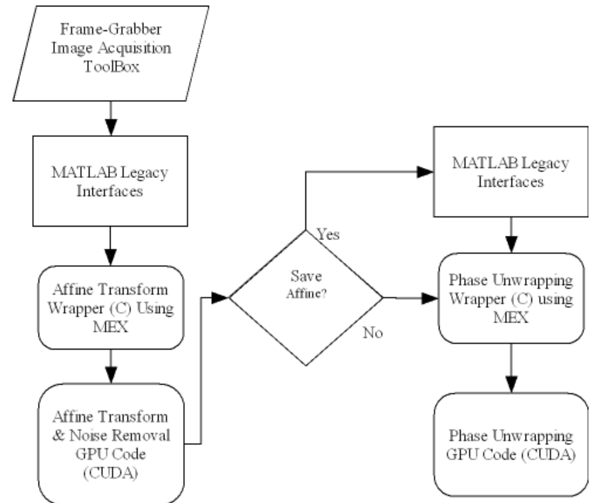

Figure 4: The flow of the GPU processing in OQM System.

To accommodate working with Matlab, we use the NVMex script provided by NVIDIA [11] that allows us to compile CUDA code and make it callable from within Matlab. We utilize Matlab's External Interface (MEX) [1] which can be used to generate dynamically-linked subroutines from C code. NVmex is a script from NVIDIA that uses MEX and NVCC in order to make it possible to call CUDA code from Matlab in a manner similar to regular C code.

6.1 Usage of MEX Files

The only extra overhead associated with using MEX files is to include a simple gateway function that contains function calls needed to make arrays within Matlab (i.e., mxArrays) accessible to our wrapper. This allows us to send data generated by Matlab to the GPU by simply using pointer passing.

To use MEX files, we wrote wrappers for the GPU-based code. The wrapper contains a function call to the code that communicates with the GPU. Our code for unwrapping or affine transforms did not need to be changed. The flow of the system is as shown in Figure 4. Once the image is acquired by the frame-grabbers and we move into the MEX wrapper, all disk IO associated with saving state is eliminated and there is no need to return to Matlab.

6.2 Image Examples

Next, we provide examples using typical image data sets and discuss how they are used. The camera frame-grabbers capture the magnitude of all four different kinds of images. The images shown here are of a mouse embryo with four cells. The four types of images (mixed, signal only, reference only and dark) are shown in Figures 5to 8 and are obtained for each camera by manipulating the shutters. This equals a total of 16 images (4 types of images, 4 cameras). We show only a single set of images from one camera.

These images undergo the Affine Transform (Section 3.1) and dark subtraction (Section 3.2) on the GPU. The signal and reference images in Figures 6 and 7 are subtracted as per equation 10 and combining the images from all four cameras we obtain the a matrix of complex numbers. The angle of this complex number yields the wrapped phase between $-\pi$ and $+\pi$, as shown in Figure 9.

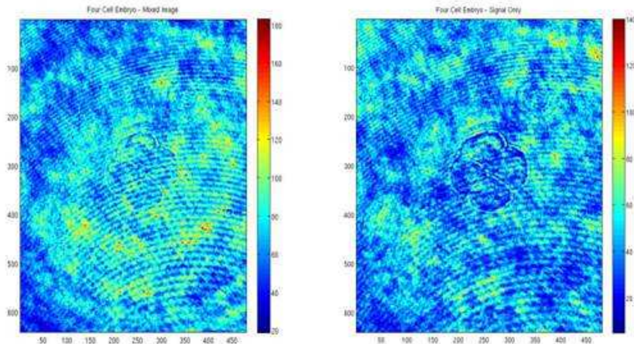

Figure 5: Mixed Image

Figure 6: Signal Image

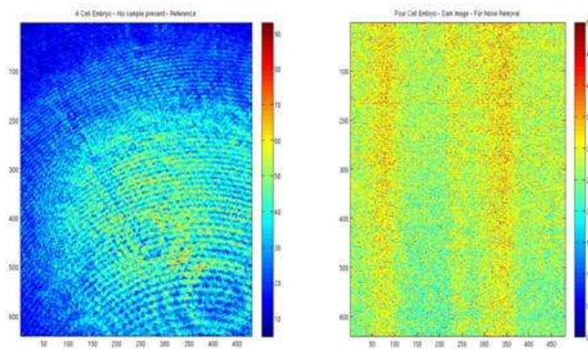

Figure 7: Reference Image

Figure 8: Dark Image

As seen, the image in Figure 9 has transitions and discontinuities around the edges of the sample (i.e., the changes in color around the border between the sample and the media). It is not easy to discern details of the sample due to the multiple small transitions inside the sample. However, after completing the unwrap, we can see in Figure 10 the color transition from the culture media to the middle of the sample is very smooth and even an untrained eye can see difference in the level of detail in the embryo. It is fairly easy to count the 4 cells seen in the embryo in Figure 10. This final image is the motivation and output of this research.

7. PERFORMANCE & RESULTS

Next, we discuss the performance of our accelerated application on the GPU.Due to the complexity and heterogeneous nature of our application, the speedup and performance of our system is quoted not only for our GPU-based kernels, but for the complete system which includes the time spent in Matlab wrappers and the disk IO.

7.1 Phase Unwrap Only

The frame grabbers used in the Optical Science Lab acquire images at a resolution of 640*480. To produce a reasonable baseline, we

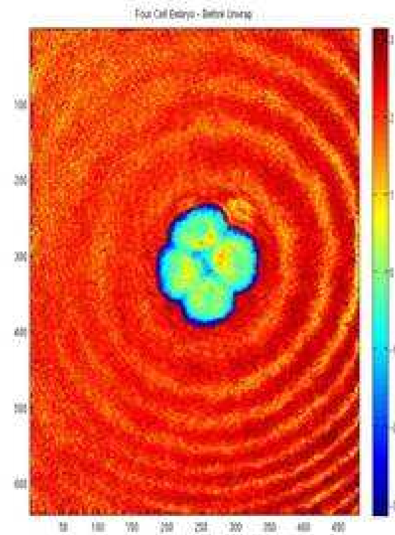

Figure 9: Wrapped Phase Image

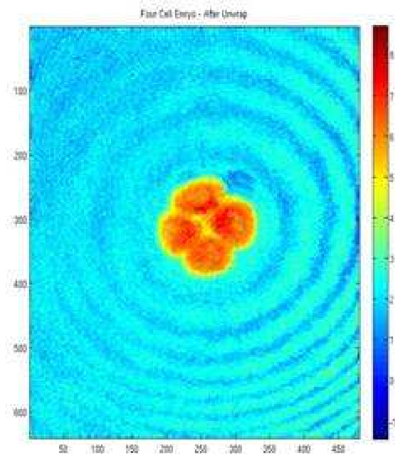

Figure 10: Unwrapped Image

use a mature C-based implementation of phase unwrap described in [4]. The implementation provided [2] used the FFTW library [3] which is a highly-tuned library for implementing an FFT on a regular CPU. In the phase unwrapping problem, due to the complexity and heterogeneous nature of the system, we analyze the main parts of the algorithm and look at the individual speedups rather than cumulative performance numbers.

Execution time is reduced from an average of 15 seconds for the software FFTW based version to 2.8 seconds on the GPU. This includes the time to run the preconditioner and perform the conjugate gradient calculations. These numbers were generated for a glass bead dataset and represent 200 iterations of the PCG kernel. The glass bead in oil is an excellent benchmark to use since it produces a very large number of residues and is usually very hard to unwrap.

More detailed runtimes are provided in Table 7.1. The MEX+GPU column uses the same implementation as the GPU. However the code is called from within Matlab as discussed in Section 6. We do not quote a speedup number for the MEX+GPU IO activity since the only IO related activity is simple pointer passing.

Figure 11 shows a breakdown of the percentage of time taken by each component of the unwrapping code. An important detail to be noted is that the "IO Activity" term in Table 7.1 defines "IO Activity" as memory accesses done when reading and writing the phase data. In both the baseline and the GPU implementation it is the same value since both read and write the same data to disk. So the percentage of time spent on I/O activity increases in Figure 11 as can bee seen in Table 7.1 for the GPU case. The results for the MEX+GPU case in Table 7.1 indicate that we are not bound by disk activity. Only .02 seconds is spent in the MEX wrapper to process data visible from within MATLAB and pass the pointers to the CUDA code.

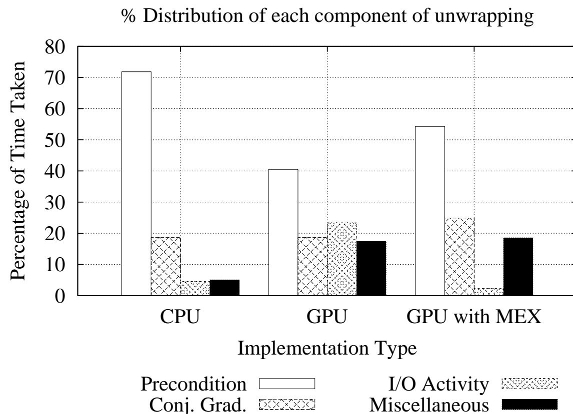

Figure 11: Time taken by each component for C implementation and GPU implementation.

7.2 Affine Transforms

Run time of the Affine Transform algorithm, unlike the Phase Unwrap algorithm, remains the same irrespective of the type of data used. In this case the baseline performance was the Matlab implementation of an Affine Transform. The Affine Transform was called after the Image Acquisition Toolbox's getdata() function is run. The baseline time taken by a Matlab implementation was 1.25 seconds. The GPU implementation (even using the unoptimized scatter operation described in Algorithm 3) completes the Affine Transform processing and the BLAS calls for noise removal in 0.2seconds, thus yielding a 6.25X speedup

8. FUTURE WORK

Future work on this framework includes adding support for multiple GPUs which would allow us to also implement a coarser grain of parallelism (e.g., running multiple unwrap algorithms). Such work would be useful for high throughput imaging and broaden the scope of OQM to new domains where we may need to acquire multiple images in less time than it takes to process a single image.

We can also explore using cudastreams, which would allow us to overlap execution and computation. The code of the Affine Transform and the Phase Unwrap are both good targets for further optimization and we can improve their performance even more. Pursuing better methods to implement scatter/gather would also be helpful since they are used in both affines and preconditioning.

9. CONCLUSION

We have implemented the Minimum L^p Norm Phase Unwrap algorithm on CUDA. In addition we implemented parallelized algorithms for Affine Transform and noise removal on a GPU. We obtain a speedup of 7.3X for the entire system by accelerating not only GPU code, but by also reducing the overhead associated with interfacing with Matlab by reducing some disk activity. Our approached does not change the image acquisition methodology. Using a GPU, we are closer to our goal of real-time OQM imaging. We plan to pursue using multiple GPUs in future work.

10. ACKNOWLEDGMENTS

We would like to acknowledge the assistance of the members of the Optical Science Laboratory at Northeastern University. This work was supported in part by CenSSIS, the Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the NSF (Award Number EEC-9986821), by the Institute of Complex Scientific Software at Northeastern University, by The MathWorks, and by equipment donations from NVIDIA.

11. REFERENCES

[1] MATLAB External Interface Guide, 2007.

[2] C. Smith. Phase Unwrapping Algorithms, Master's Project, Northeastern University, 2004.

[3] Frigo, M. and Johnson, S.G. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb. 2005.

[4] D. Ghiglia and M. Pritt. *Two-dimensional phase unwrapping: theory, algorithms, and software*. Wiley New York:, 1998.

[5] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[6] D. O. Hogenboom, C. A. DiMarzio, T. J. Gaudette, A. J. Devaney, and S. C. Lindberg. Three-dimensional images generated by quadrature interferometry. *Opt. Lett.*, 23(10):783–785, 1998.

[7] A. K. Jain. *Fundamentals of digital image processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[8] P. Karasev, D. Campbell, and M. Richards. Obtaining a 35x Speedup in 2D Phase Unwrapping Using Commodity

Table 3: CUDA Implementation versus CPU based C Implementation (using FFTW)

	Baseline Time(sec)	GPU Time(sec)	GSpeedupdUp	MEX & GPU Time(sec)	MEX & GPU Speedup
Preconditioning	11.17	1.2	9.3X	1.2	9.3X
Conjugate Grad.	2.89	0.55	5.25X	0.55	5.25X
IO Activity	0.7	0.7	1X	0.02	NA
Miscellaneous	0.79	0.51	1.53X	0.41	1.92X
Total	15.55	2.97	5.24X	2.16	7.20X

Graphics Processors. *Radar Conference, 2007 IEEE*, pages 574–578, April 2007.

[9] J. Makhoul. A fast cosine transform in one and two dimensions. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 28(1):27–34, Feb 1980.

[10] Mark Harris. Optimizing Parallel Reduction in CUDA. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf, Last accessed December 2008.

[11] NVIDIA. *Accelerating MathWorks MATLAB with CUDA*, 2007.

[12] NVIDIA. *CUFFT Library*, 2007.

[13] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[14] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[15] Sherman Bragranza. Master's Thesis, 2008. Northeastern University, http://www.ece.neu.edu/groups/rcl/publications.html.

[16] W. C. Warger, II, J. A. Newmark, C. Chang, D. H. Brooks, C. M. Warner, and C. A. DiMarzio. Combining optical quadrature and differential interference contrast to facilitate embryonic cell counting with fluorescence imaging for confirmation. In D. V. Nicolau, J. Enderlein, R. C. Leif, D. L. Farkas, and R. Raghavachari, editors, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 5699 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pages 334–341, Mar. 2005.