Off-axis quantitative phase imaging processing using CUDA: toward real-time applications

Hoa Pham,[1,*] Huafeng Ding,[1] Nahil Sobh,[2] Minh Do,[1] Sanjay Patel,[1] and Gabriel Popescu[1]

[1]Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA, and Beckman Institute for Advanced Science & Technology, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
[2]National Center for Supercomputing Applications, Department of Civil and Environmental Engineering, and Department of Mechanical Engineering and Sciences, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
**hoapham2@illinois.edu*

Abstract: We demonstrate real time off-axis Quantitative Phase Imaging (QPI) using a phase reconstruction algorithm based on NVIDIA's CUDA programming model. The phase unwrapping component is based on Goldstein's algorithm. By mapping the process of extracting phase information and unwrapping to GPU, we are able to speed up the whole procedure by more than 18.8× with respect to CPU processing and ultimately achieve video rate for mega-pixel images. Our CUDA implementation also supports processing of multiple images simultaneously. This enables our imaging system to support high speed, high throughput, and real-time image acquisition and visualization.

OCIS codes: (180.3170) Interference microscopy; (170.6920) Time-resolved imaging; (100.5070) Phase retrieval; (100.5088) Phase unwrapping.

References and links

1. G. Popescu, in *Methods in Cell Biology*, B. P. Jena, ed. (Academic Press, San Diego, 2008), pp. 87–115.
2. C. Depeursinge, "Digital holography applied to microscopy," in *Digital Holography and Three-Dimensional Display*, T.-C. Poon, ed. (Springer, 2006), p. 98.
3. D. C. Ghiglia and M. D. Pritt, *Two-Dimensional Phase Unwrapping: Theory, Algorithms, and Software* (Wiley, New York, 1998).
4. G. Popescu, T. Ikeda, R. R. Dasari, and M. S. Feld, "Diffraction phase microscopy for quantifying cell structure and dynamics," Opt. Lett. **31**(6), 775–777 (2006).
5. D. Kirk and W.-m. Hwu, *Programming Massively Parallel Processors Hands-on With CUDA* (Morgan Kaufmann, Burlington, MA, 2010).
6. P. A. Karasev, D. P. Campbell, and M. A. Richards, "Obtaining a 35x speedup in 2d phase unwrapping using commodity graphics processors," in *2007 IEEE Radar Conference* (IEEE, 2007), pp. 574–578.
7. P. Mistry, S. Braganza, D. Kaeli, and M. Leeser, *Accelerating Phase Unwrapping and Affine Transformations for Optical Quadrature Microscopy Using CUDA* (ACM, Washington, D.C., 2009), pp. 28–37.
8. G. Popescu, L. P. Deflores, J. C. Vaughan, K. Badizadegan, H. Iwai, R. R. Dasari, and M. S. Feld, "Fourier phase microscopy for investigation of biological structures and dynamics," Opt. Lett. **29**(21), 2503–2505 (2004).
9. T. Ikeda, G. Popescu, R. R. Dasari, and M. S. Feld, "Hilbert phase microscopy for investigating fast dynamics in transparent systems," Opt. Lett. **30**(10), 1165–1167 (2005).
10. Y. K. Park, C. A. Best, K. Badizadegan, R. R. Dasari, M. S. Feld, T. Kuriabova, M. L. Henle, A. J. Levine, and G. Popescu, "Measurement of red blood cell mechanics during morphological changes," Proc. Natl. Acad. Sci. U.S.A. **107**(15), 6731–6736 (2010).
11. Y. K. Park, M. Diez-Silva, G. Popescu, G. Lykotrafitis, W. Choi, M. S. Feld, and S. Suresh, "Refractive index maps and membrane dynamics of human red blood cells parasitized by *Plasmodium falciparum*," Proc. Natl. Acad. Sci. U.S.A. **105**(37), 13730–13735 (2008).
12. H. F. Ding, Z. Wang, F. Nguyen, S. A. Boppart, and G. Popescu, "Fourier transform light scattering of inhomogeneous and dynamic structures," Phys. Rev. Lett. **101**(23), 238102 (2008).
13. NVIDIA, *NVIDIA CUFFT Library*.
14. NVIDIA, *NVIDIA CUDA Programming Guide 3.2* (2010).

#145283 - $15.00 USD Received 14 Apr 2011; revised 23 May 2011; accepted 30 May 2011; published 1 Jun 2011

1. Introduction

In the past decade, quantitative phase imaging (QPI) has attracted increasing scientific interest in the area of cell and tissue imaging as it can study structure and dynamics with nanoscale sensitivity and without exogenous contrast agents [1,2]. Typically, in order to obtain the pathlength map from an acquired interferogram image, QPI involves off-line post processing. In particular, off-axis methods require an unwrapping algorithm to remove the high-frequency spatial modulation. Phase unwrapping is the process of reconstructing the true phase information from the measured wrapped values which are between $-\pi$ to $+\pi$. High throughput, high speed, real-time phase unwrapping is highly desirable in many applications including applied physics and biomedicine. However, to the best of our knowledge, currently there are no phase unwrapping algorithms that allow QPI operation at video rates (i.e., ~30 frames/s).

There are two main types of phase unwrapping algorithms: path-following algorithms and the minimum-norm algorithms [3]. We chose Goldstein's branch cut method, which is a classic path-following algorithm and allows for high throughput. Although Goldstein's algorithm is the fastest phase unwrapping algorithm, its implementation in sequential code is still slow and far from meeting the real-time requirements. For example, for our *diffraction phase microscope* [4], C-code Goldstein's algorithm takes about 150 milliseconds to unwrap a 1024x1024 phase image, in addition to about 300 milliseconds to extract phase image from the raw microscopy data. The total processing time is about half a second which is much larger than our 30 ms target.

In the CUDA programming environment, graphics processing units (GPUs) can be regarded as computation devices operating as coprocessors to the central processing unit (CPU) [5]. The idea is to process computationally-intensive parts in parallel by using multiple computation units. The CUDA architecture consists of hundreds of processor cores that operate together to process different segments of the data set in the application. Previous work on using GPUs for 2D phase unwrapping include Karasev et al [6] and Mistry et al [7]. The former implemented a weighted least squares algorithm for Interferometric Synthetic Aperture Radar (IFSAR) data and the latter implemented a Minimum L^p norm phase unwrap algorithm for optical quadrature microscopy system. However, due to the computational complexity of minimum norm algorithms, the processing time of these algorithms is still too large and far from satisfying real-time requirements.

In this paper, we develop an unwrapping algorithm based on Goldstein's algorithm using CUDA to achieve real-time requirements. The reasons to choose Goldstein's algorithm are twofold. First, it is the fastest algorithm and can potentially be improved with CUDA implementation to achieve real-time requirements. Secondly, it performs effectively and satisfactorily for our targeted QPI system. The rest of the paper is organized as follow: Section 2 briefly introduces background information on our off-axis QPI system. Section 3 describes steps involved in the phase reconstruction process of the QPI imaging method. Section 4 illustrates our proposed CUDA-based phase unwrapping algorithm. In section 5, we present performance results of the proposed algorithm. Section 6 includes the conclusion and our future works. Finally, the Appendix provides information about CUDA GPU architecture, a summary of Goldstein's algorithm and details of our CUDA implementation of residue identification algorithm.

2. Off-axis QPI

Off-axis interferometry takes advantage of the *spatial phase modulation* introduced by the *angularly shifted* (tilted) reference plane wave and the spatially-resolved measurement allowed by a 2D detector array such as a CCD (Fig. 1). Essentially off-axis interferometry is the spatial equivalent of heterodyne detection in the time domain. Compared to phase-shifting methods (see, e.g., [8]), off axis-interferometry allows for single shot measurements and, thus, fast acquisition rates.

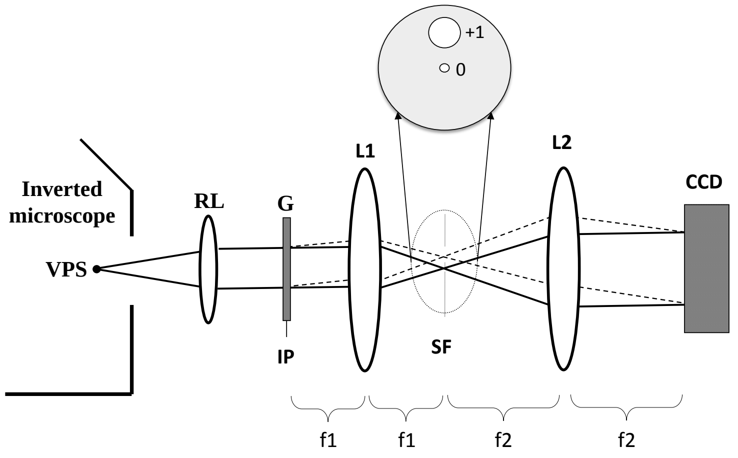


Fig. 1. Diffraction phase microscope: VPS virtual point source, RL relay lens, G grating, IP image plane, SF spatial filter, L1-2 lenses, CCD charged coupled device.

The intensity distribution of the interferogram at the detector plane takes the form (in the absence of noise)

$$I(x,y)=|U_i(x,y)|^2+|U_r|^2+2|U_r|\cdot|U_i(x,y)|\cdot\cos\left[\Delta k\cdot x+\phi(x,y)\right]. \tag{1}$$

The goal is to isolate the term $\cos\left[\Delta k\cdot x+\phi(x,y)\right]$ from the measurement and then numerically compute its sine counterpart via a Hilbert transform. In order to achieve this, $|U_i|$ and $|U_r|$ can be independently measured by blocking one beam of the interferometer and measuring the resulting intensity of the other. As a result, the *cosine* term is obtained by itself, which can now be interpreted as the real part of a (spatial) complex analytic signal.

The corresponding imaginary part is further obtained via a Hilbert transform, as [9]

$$\sin\left[\Delta k\cdot x+\phi(x,y)\right]=P\int\frac{\cos\left[\Delta k\cdot x'+\phi(x',y)\right]}{x-x'}dx, \tag{2}$$

where P indicates the principle value integral. Finally, the argument of the trigonometric functions is obtained uniquely as

$$\phi(x,y)+\Delta kx=\arg\left[\cos(\Delta kx+\phi),\,\sin(\Delta kx+\phi)\right]. \tag{3}$$

Importantly, the 2D phase map can be retrieved via a single CCD exposure. The main challenge is to produce a stable interferogram, i.e., maintain a stable phase relationship between the reference and the sample field. Diffraction phase microscopy (DPM) [4] is a QPI technique that combines the single shot feature of off-axis methods with the stability of common path interferometry and thus, renders highly sensitive phase images with high throughput. With these features, recently DPM has enabled new biomedical studies [10–12].

Our DPM experimental setup is shown in Fig. 1. The second harmonic radiation of a Nd:YAG laser (λ = 532nm) is used as illumination for an inverted microscope, which produces the magnified image of the sample at the output port. The microscope image appears to be illuminated by a virtual source point VPS. A relay lens RL collimates the light originating at VPS and replicates the microscope image at the plane IP. A diffraction phase grating G (hence "*diffraction phase microscopy*") is placed at this image plane and generates multiple diffraction orders containing full spatial information about the image. We select two diffraction orders (0th and 1st) that can be further used as reference and sample fields in a very compact Mach-Zehnder interferometer. In order to accomplish this, a standard spatial filtering lens system, L_1-L_2, is used to select the two diffraction orders and generate the final interferogram at the CCD plane. The 0th order beam is low-pass filtered using the spatial filter SF positioned in the Fourier plane of L_1, such that at the CCD plane it approaches a uniform field. The spatial filter allows passing the entire frequency content of the 1st diffraction order beam and blocks all the other orders. The 1st order is thus the imaging field and the 0th order plays the role of the reference field. The two beams traverse the same optical components, i.e. they propagate along a common optical path, thus significantly reduces the longitudinal phase noise.

3. Phase reconstruction

In this section, we describe the steps involved in the phase reconstruction process in off-axis QPI systems. Before the unwrapping process can be started, phase information needs to be extracted from interferograms captured from the cameras. Figure 2 illustrates the phase reconstruction procedure in QPI system.

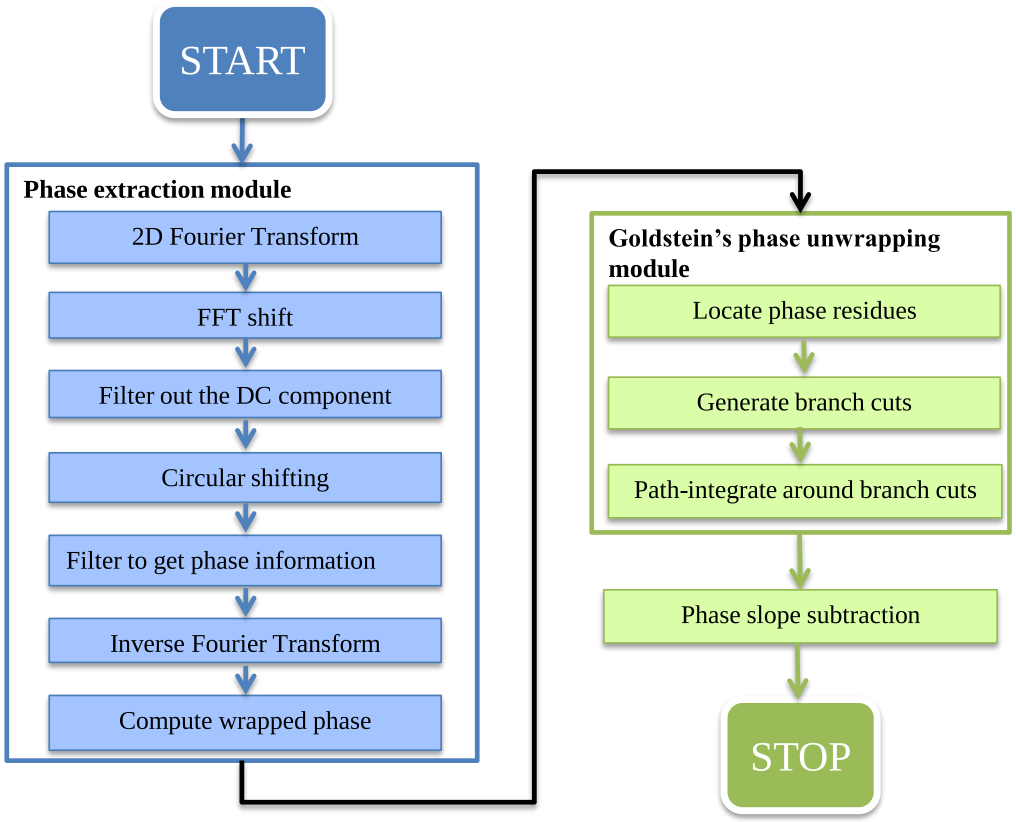


Fig. 2. Phase reconstruction in QPI system

3.1. Phase extraction from interferograms

This module is to extract the phase information from the interferograms (Fig. 3a). The Hilbert transform of a function [Eq. (2)] is equivalent with the following combination: Fourier transform to frequency domain, followed by multiplication by a step function in the frequency domain and inverse Fourier transform to real space. First, an image captured from the cameras is transformed to the spatial frequency domain using two-dimensional Fourier transform and then shifted to move the zero-frequency component to the center of the image. This shifting basically swaps the first quadrant with the third and the second quadrant with the fourth. The power spectrum of the shifted image, as illustrated in Fig. 3b, contains three peaks with the middle one located at the center of the image containing the DC component (the fringes of the interferogram) and the other two first-order peaks contain the same information about the phase. In the next step, we will need to find the coordinates locating one of these two peaks. These coordinates will be used to design filters to remove the DC component as well as to extract the phase information of the image. The circular shifting step is to shift the first-order peak to the center of the image, which will then be extracted in the next step. Finally, we apply inverse FFT and compute the phase values using the *arg* function.

3.2. Phase Unwrapping

The phase of interest, φ, is wrapped to the range (-π, π] such that measured function is:

$$\psi(i) = \varphi(i) + 2\pi k(i), \tag{4}$$

where *k(i)* is an integer such that $-\pi < \psi \leq \pi$ and *i* is the array index. The phase unwrapping problem is to find an estimate $\phi(i)$ for the actual phase $\varphi(i)$ from the measured $\psi(i)$.

In 1D, assuming that the true phase has local gradient less than π radians, the wrapped version of the image can be unwrapped by the following Itol's method [3]:

Step 1: Compute the phase differences: $D(i) = \psi(i+1) - \psi(i)$, for i = 0,…, N-2, with N the length of the array.

Step 2: Compute the wrapped phase differences: $\Delta(i) = \arctan\{\sin D(i), \cos D(i)\}$, for I = 0,…, N-2.

Step 3: Initialize the first unwrapped value: $\phi(0) = \psi(0)$.

Step 4: Unwrap by summing the wrapped phase differences: $\phi(i) = \phi(i-1) + \Delta(i-1)$, for I = 0,…, N-2.

Phase unwrapping becomes much more complicated in the 2D space. When noise is present, phase gradients may be greater than π causing the presence of residues and leading to image corruption.

#145283 - $15.00 USD Received 14 Apr 2011; revised 23 May 2011; accepted 30 May 2011; published 1 Jun 2011

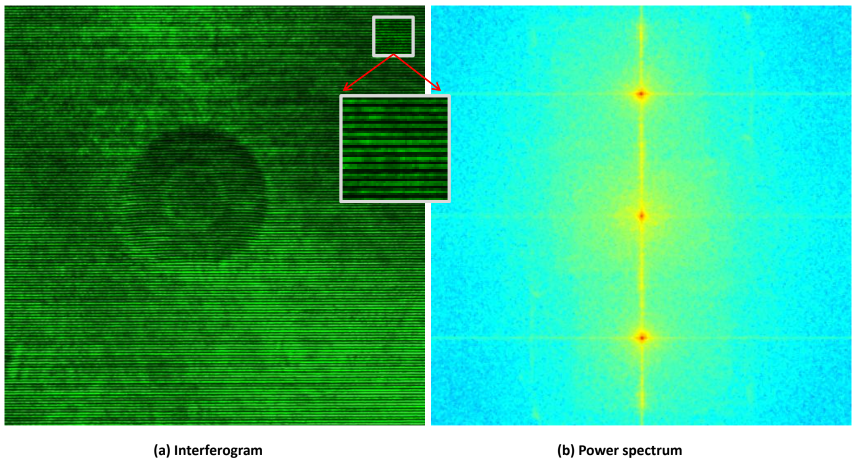


Fig. 3. Interferogram of a red blood cell (a) and its power spectrum (b)

A pixel is called a *residue* if the integral over a closed four pixel loop is not zero. We have the following residue theorem [3] for two-dimensional unwrapping:

$$\int \nabla \varphi(\mathbf{r})\,\mathbf{dr} = 2\pi q, \tag{5}$$

where q is the total *residue charges*.

Therefore, if the residue charges are balanced by connecting residues of opposite charges with branch cuts, the unwrapping process gives consistent results along any path which does not cross branch cuts. We use Goldstein's algorithm for phase unwrapping. Details of Goldstein's algorithm are provided in Appendix B.

The last step in the algorithm is to subtract the horizontal phase slope from the image to calibrate the tilting of the specimen plane. We first use least square fitting to find the coefficients of the plane representing the slope of the tilting. This step is very computational. Fortunately, these tilting coefficients only need to be computed for the first image during one measurement. For the successive images, we can use the same coefficients to subtract the tilting plane point by point since it does not change during one measurement.

4. CUDA implementation

In this section, we present the implementation of the phase reconstruction algorithm on GPU. To further utilize the GPU power, we developed a program that can process multiple images simultaneously. This will be useful for streaming-type applications.

4.1. Phase extraction module

The process of extracting phase information from interferogram images includes Fourier transforms (one for forward and one for inverse direction), building the filters, point-wise matrix multiplications (for filtering), shifting of matrix entries and computing the argument of the complex number of every pixel value of the image. All of these steps are quite computationally intensive in sequential C code and can be implemented in CUDA very efficiently.

In QPI, since the coordinates of the first-order maxima and thus the filters remain the same for one measurement, we only need to find these coordinates and compute the filters once for the first image and use those computed filters to process successive images.

For Fourier transforms, we use the CUFFT library developed by NVIDIA [13]. The CUFFT library provides functions to rapidly compute 1D, 2D and 3D Fourier transforms in CUDA C. The filtering part is just point-wise matrix multiplication in the frequency domain and very straightforward for parallel implementation since there are no dependencies between data points. This is implemented using a thread-per-pixel model which assigns a thread to compute for each pixel of the image. Likewise, FFT shifting swaps the first with the third quadrants and the second with the fourth quadrants of the image, and thus is highly parallel and straightforward to be implemented in CUDA. Each thread will be responsible for swapping one pair of pixels from the first and the third quadrants and one pair of pixels from the second and the fourth quadrants.

In the circular shifting step, we need to circularly shift all the columns and then rows of pixels by some shift strides. The shift strides in vertical and horizontal directions of the image are determined by the coordinates of the first-order peak in the power spectrum as mentioned in section 3.1. We first copy all the pixel values in the image array to a buffer array and then each thread copies a pixel value in the buffer array to a corresponding shifted pixel in the original image array. Finally, the last step is to compute the wrapped phase value at each pixel by using arctangent function. This computation is also very straightforward for parallelization and again each thread computes the phase value for one pixel. These values are stored in an array, called a phase array.

For all the steps in this module, we use a grid of 16 × 16 thread blocks. The number of block is determined by the size of the image and the number of images we want to process simultaneously.

4.2. Residue identification

Residue identification is done based on a pixel by pixel basis. A pixel is a positive (negative) residue if the integral over a closed four pixel loop is greater (smaller) than zero. Therefore, it is also straightforward for parallelization; each thread computes an integral over a closed four pixel loop to decide if the pixel top left corner of the loop is a residue or not. This information is stored in an array of byte type, called bitflag array. Further, for each image we use one integer-typed variable to store the number of residues and one array to store positions of those residues. This information is for later use in the branch cut placement step. Details of bitflag structure and residue identification algorithm are discussed in Appendix C.

4.3. Branch cut placement

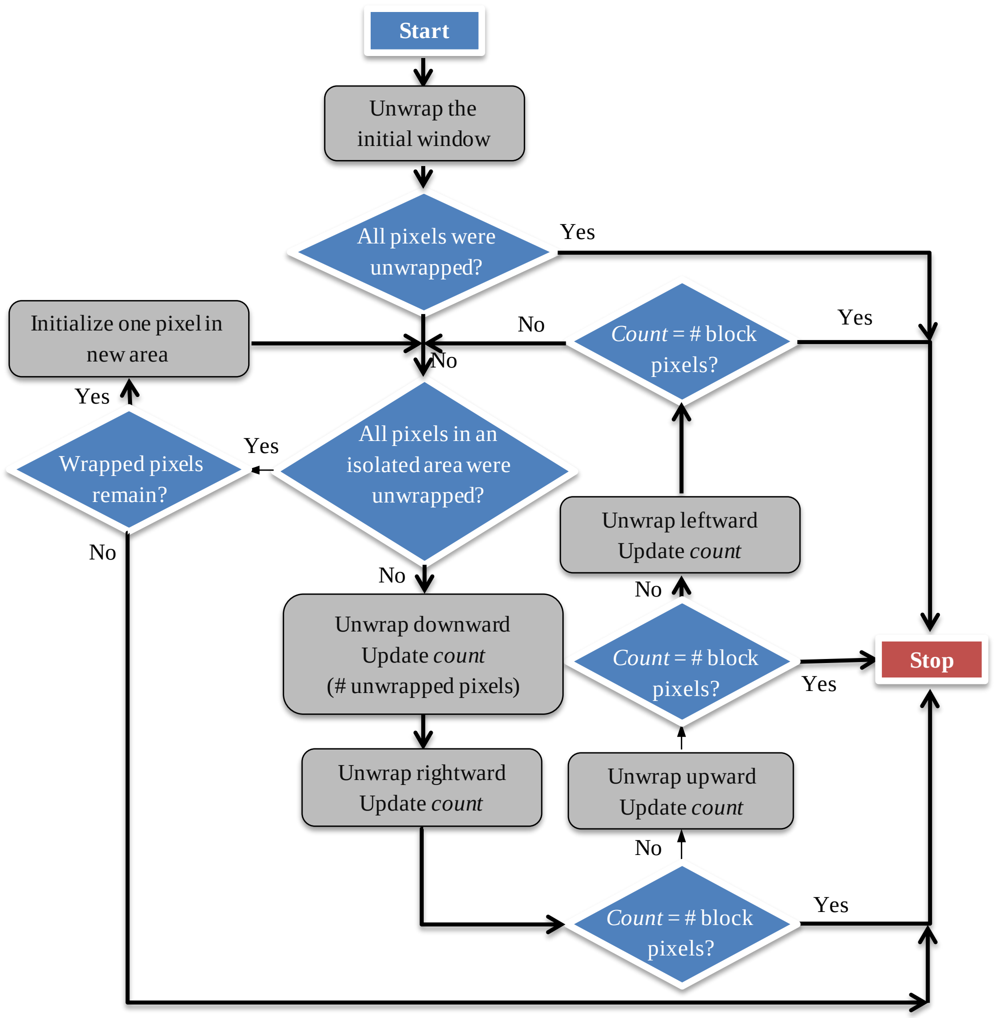
The branch cut algorithm includes processes of enlarging and searching over a search box on the image and the charge is cumulatively computed. This process requires information about other residues it encounters during the search, such as whether a residue is branch cut pixel or it has already been connected to some other residues, etc. Thus, this process cannot be parallelized for CUDA implementation. One way to solve this problem is to implement this part in the host code. However, in order to do this, we have to copy the data from the GPU device's memory to the host and then copy it from the host back to the device after the branch cut placement has been done, which introduces a significant time delay.

In order to avoid this back and forth copying, we implement this step in CUDA by using one thread to process the branch cuts for each image. Instead of scanning over the whole image as in the original Goldstein's algorithm, we only scan the residue lists stored in the residue identification step. This way of implementation may be computationally intensive if the number of residues is large, resulting in longer running time compared to the implementation of Goldstein's branch cut algorithm in the host code. For our targeted applications in optical phase imaging, the number of residues is often small and this implementation fits well and indeed performs much better than the host code one.

4.4. Unwrapping around branch cuts

This step is also difficult to be parallelized, since unwrapping a pixel requires that one of its neighbors has been already unwrapped. The most direct way to do this in CUDA is that each thread performs the computation for one pixel by first checking the pixel's neighbors and then unwrapping the pixel if one of the neighbors is unwrapped. After that, all the threads must be synchronized to update the pixels' status before repeating the process of checking and unwrapping. This process stops when all the pixels are unwrapped. However, we know that only threads in the same block can cooperate, and the maximum number of threads per block supported by existing devices is 1024, limiting us to very small images. Furthermore, each iteration needs to read data of the neighboring pixels, thus when the size of the image gets larger, this process may become computationally intensive due to repeated memory reading.

An alternative way is to use each thread to unwrap one column of the image and scan through pixels in each column. If any of a pixel's neighbors is flagged as unwrapped, that pixel will be unwrapped and then flagged as such. The process will be repeated until all possible pixels are unwrapped. However, this process requires data from four neighbors for each pixel. This causes large demand on global memory access and may slow down the program. Instead, we choose to change the direction of scanning after each scan and in each scan, we only check the previous pixel in the scanning direction to see if it is flagged as unwrapped. Specifically, we use one thread to unwrap each row or column of the image. First, we scan the image from top to bottom, then from the left to the right, then from bottom to top



Start
Unwrap the initial window
All pixels were unwrapped?
Yes
Initialize one pixel in new area
No
Count = # block pixels?
Yes
No
Yes
Yes
Wrapped pixels remain?
All pixels in an isolated area were unwrapped?
Unwrap leftward Update count
No
No
No
Unwrap downward Update count (# unwrapped pixels)
Count = # block pixels?
Stop
Yes
Unwrap rightward Update count
Unwrap upward Update count
No
Yes
Count = # block pixels?

Fig. 4. Flow chart of unwrapping algorithm

and then from right to left. The scanning continues until no more pixels can be unwrapped. Then, either all the pixels have been unwrapped or there is an isolated region in the image. For each isolated region, we first initialize a pixel in the region and repeat the above process.

We now discuss in more detail our unwrapping algorithm. We first unwrap pixels in a small window of pixels at the center of the image by repeating the process of checking and unwrapping as mentioned above. In our implementation, we choose an initial window of size of 32 × 32 pixels. After the initial window has been unwrapped, we divide the image into four regions; each region will be unwrapped by a block of threads. In each block, we use two loops of scanning and changing scanning directions as described above. The inner loop is to process pixels in one isolated area and the outer loop is to handle all isolated areas. After each scanning, if the number of unwrapped pixels equals the total number of pixels in the block, the scanning process stops. Figure 4 presents a flow chart of the unwrapping algorithm.

In the last step, we need to subtract the phase slope from the image. As mentioned in Section 3.2, once the tilting coefficients are computed for the first image, we can use those coefficients to calibrate for the tilting plane from the successive images. Since this tilting plane subtraction procedure can be done point by point, it can be implemented very efficiently in CUDA by letting one thread to compute the adjusted phase value for each pixel. Therefore, this step can be done very fast in our CUDA implementation.

5. Performance results

In this section, we discuss the performance of our algorithm on GPU. We tested the algorithm on a Windows machine with Intel® Core i5 CPU with clock rate of 3.2 GHz and 8 GB RAM memory. We use NVIDIA® GeForce® GTX 470M GPU which supports CUDA programming.

Figure 5 illustrates an example of a red blood cell imaged by DPM. Figure 5a shows the wrapped phase image associated with the interferogram in Fig. 3a after the phase extraction. Figure 5b shows the result after the unwrapping procedure. The vertical and horizontal axes illustrate pixels' positions in 2D image and the color bar denotes the phase values before and after unwrapping.

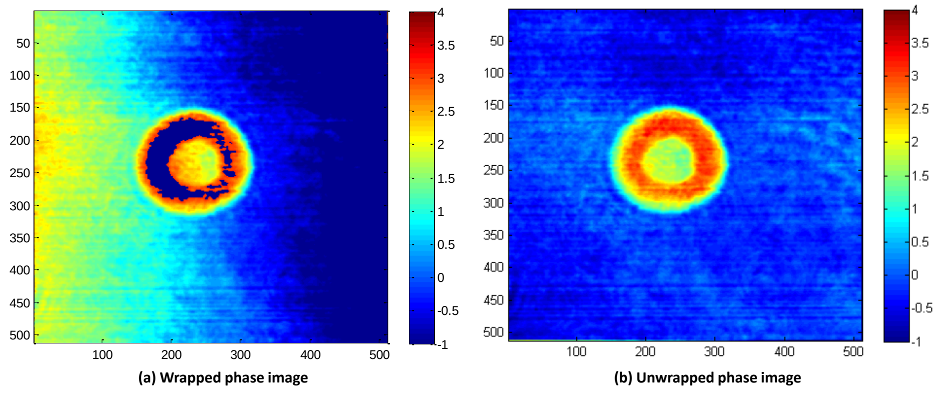


Fig. 5. Phase unwrapping of red blood cell image

For comparison, we also implemented a C-code based program and use the unwrap algorithm as described in [3]. For the phase extraction part, we use FFTW library to compute Fourier transforms. Our GPU version is implemented in CUDA C, and compiled with the Microsoft Visual Studio compiler.

Table 1 compares the run time between the two implementations. The results shown were averaged over 20 images for each image size. As mentioned earlier, our program supports multiple image frames. For sequential C-code program, the run time for multiple frames simply scales linearly with the number of frames. For GPU implementation, the total run time includes time for the memory copy of interferogram from host to device and also a memory copy of output unwrapped image from device to host.

Table 1. CUDA implementation versus C based sequential implementation

Image Size	CPU/GPU	Phase extraction (ms)	Residue Identification (ms)	Branch cut Placement (ms)	Unwrap (ms)	Total (ms)
1024 × 1024	CPU	317.42	43.42	6.74	89.32	460.7
1 frame	GPU	5.05	0.58	1.125	10.014	24.55
	Speedup factor	**62.86**	**74.19**	**5.99**	**8.92**	**18.77**
1024 × 1024	CPU	3174.2	434.2	67.4	893.2	4607.4
10 frames	GPU	40.486	5.55	1.128	45.285	111.1
	Speedup factor	**78.4**	**78.19**	**59.71**	**19.72**	**41.47**
512 × 512	CPU	71	11	5	16	105
1 frame	GPU	2.18	0.2	0.02	1.87	8
	Speedup factor	**32.61**	**55.84**	**250**	**8.55**	**13.13**
512 × 512	CPU	710	110	50	160	1050
10 frames	GPU	11.57	1.4	0.02	6.722	26
	Speedup factor	**61.37**	**78.57**	**2500**	**23.8**	**40.38**

Clearly, the GPU implementation demonstrates tremendous improvement on run time performance. The total run time for a single 1024 × 1024 image reduced from an average of 460 milliseconds for the sequential C-code implementation to 24.55 milliseconds on GPU, which is now suitable for video rate. The total run time for a single lower resolution (512 × 512) image is 8 milliseconds, allowing for much higher image acquisition rates. Furthermore, we note larger speedup when multiple images are processed simultaneously which is extremely useful for streaming applications.

6. Conclusion and future work

In this paper, we presented a phase unwrapping algorithm in CUDA C. The algorithm is based on the classical Goldstein's branch cut algorithm. The algorithm demonstrates a tremendous improvement, of a factor of about 18× for single images, and 40× for 10 images, over sequential implementation. By implementing all functions in CUDA, we eliminate all intermediate memory copies between the host and the GPU device, reducing the run time. We obtained a speedup of 18.77× for images of size 1024 × 1024 and reduced a total run time to about 24 milliseconds which is suitable for real-time, high resolution phase reconstruction.

We anticipate that in the near future, from the unwrapped phase images, CUDA-based modules will compute in real-time quantitative parameters of the imaged objects, e.g., cell volumes, refractive indices, tissue morphological parameters, etc, useful for both basic biological studies and medical diagnosis.

Appendix A: CUDA computing architecture

In this section, we briefly describe NVIDIA's Compute Unified Device Architecture model (CUDA) to justify our CUDA-based phase unwrapping algorithm. CUDA GPU was designed to process thousands of threads simultaneously by underlying parallel stream processors. It consist of several streaming multiprocessors (SMs) as illustrated in Fig. 6a.

A CUDA program consists of both host code and device code. The host code is straight ANSI C code, which is used when there is little or no data parallelism and the device code is used when there is a rich amount of data parallelism. The NVIDIA C Compiler (NVCC) separates the two. The host code is straight ANSI C code and is compiled with standard C compilers. CUDA extends C by allowing developers to define C functions, called kernels, and the device code is compiled by the NVCC and executed on a GPU device. Based on single-instruction, multiple-thread (SIMT) architecture [5,14], CUDA maps a single kernel to a grid of threads to process different input data simultaneously. Threads in a grid are organized into

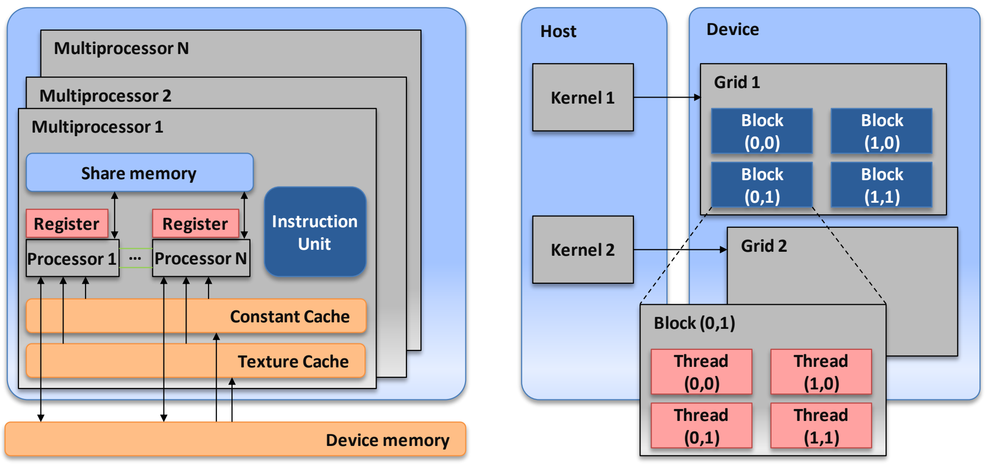


Fig. 6. CUDA GPU architecture

a two-level hierarchy, as illustrated in Fig. 6b. Threads are organized into blocks of up to three dimensions and the blocks are then organized into a one-dimensional or two-dimensional grid of thread. Each thread has its own register and each block has a shared memory which is available to all threads in that block. All threads in a block can synchronize their execution but two threads from different blocks cannot cooperate. Numbers of threads and blocks must be provided to a call of a kernel through an execution configuration. Then all threads will execute the same instruction but on different input data identified by their thread indices and block indices.

Appendix B: Goldstein's algorithm

Goldstein's algorithm includes three steps [3].

Step 1: Locate the residues

Step 2: Generate the branch cuts

Step 3: Path-integrate around the branch cuts

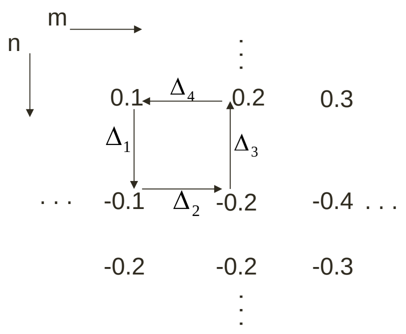


Fig. 7. Detecting residues in 2D arrays

The first step is to locate the residues in the images. For the pixel at coordinate (i,j), its residue charge will be obtained by summing the wrapped phase differences around the closed path as illustrated in Fig. 7,

$$q = \sum_{k=1}^{4} \Delta_k$$

where

$$\begin{aligned}
\Delta_1 &= W\{\psi(i, j+1) - \psi(i, j)\} \\
\Delta_2 &= W\{\psi(i+1, j+1) - \psi(i, j+1)\} \\
\Delta_3 &= W\{\psi(i+1, j) - \psi(i+1, j+1)\} \\
\Delta_4 &= W\{\psi(i, j) - \psi(i+1, j)\}
\end{aligned}$$

In the second step, the Goldstein's algorithm first scans the phase image pixel by pixel until a residue is detected. Then a 3 × 3 box is placed around the residue to search for other residues. If another residue is present, a branch cut is placed between them. If they are of opposite polarities, the charge will be balanced; otherwise the search will be continued. When another residue is found, it is connected to the residue at the center of the box whether the new residue has been connected to some other residue or not and in the latter case, its polarity will be added to the cumulative sum of the other residues. When the cumulative charge is zero, the residues are called balanced. If the cumulative charge is not zero after the search over the 3 × 3 box, the box is moved to center at each of the other residues in the previous box and searched. If the cumulative charge is still nonzero, then the box is enlarged and centered at each of the residues. This process stops when either the cumulative charge becomes zero or a border pixel is encountered.

The last step uses a flood-fill algorithm. First, a pixel is selected and its phase is stored as the unwrapped value. Then, the four neighbors of this pixel are unwrapped and the pixels are inserted into a list, called an *adjoin list*. Then, the algorithm will iteratively select a pixel from the adjoin list and unwrap and insert its neighbors into the adjoin list if those neighbors are not branch cut pixels or have not been unwrapped. Finally, when the adjoin list becomes empty, either all the non-branch cut pixels have been unwrapped or there is a region isolated by branch cuts. Isolated regions can be unwrapped independently by starting with one pixel in the region and repeating the process. The branch cut pixels are then unwrapped using their unwrapped neighboring pixels.

Appendix C: Residue identification

Table 2 illustrates the data structure of the bitflags. Bits of each byte contain binary information for one pixel of the image.

Table 2. Structure of bit flag array elements

Bit	**Information**
1st bit	POS_RES
2nd bit	NEG_RES
3rd bit	VISITED
4th bit	ACTIVE
5th bit	BRANCH_CUT
6th bit	BORDER
7th bit	UNWRAPPED
8th bit	POSTPONED

The first two bit flags of each element mark whether the pixel is positive or negative residue. The seventh bit flag marks if a pixel is unwrapped or not (in the unwrapping step). Other bit flags are used in the branch cut placement step. Additionally, for each image we use one integer-typed variable, *resNum*, to store the number of residues and one array, *res_list*, to store positions of those residues. This information is for later use in the branch cut placement step. Since threads work independently and are not synchronized, multiple threads may try to update *resNum* and *res_list* at the same time and may end up with incorrect results. In order to avoid this, we use a built-in function, called *atomicAdd*, to update the number of residues. This ensures that the operation of one thread is not interfered with by other threads. This function returns the old value of its argument, thus we can use this returned value to update our residue lists. A flow chart of residue identification function is summarized in Fig. 8. An instance of this function is run on each thread independently. The parameter *k* indicates the position of the pixel to be tested.

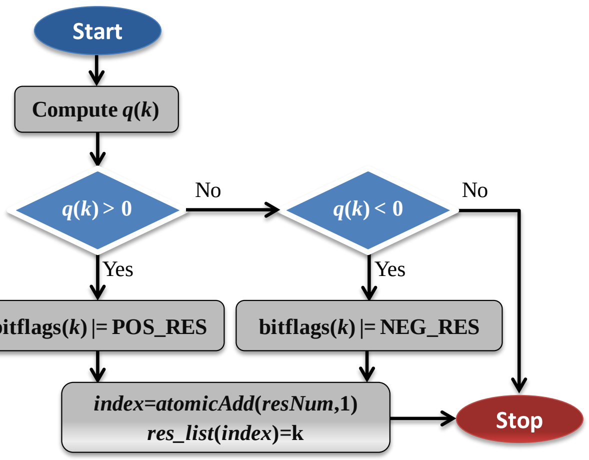


Fig. 8. Residue identification flow chart.

Acknowledgments

This research was supported in part by the National Science Foundation (grants CBET 08-46660 CAREER, CBET-1040462 MRI) and the National Cancer Institute (grant R21 CA147967-01). For more information, visit http://light.ece.uiuc.edu/.