Real-time blood flow visualization using the graphics processing unit

Owen Yang,[a,b] David Cuccia,[c] and Bernard Choi[a,b]
[a]University of California, Irvine, Beckman Laser Institute and Medical Clinic, 1002 Health Sciences Road, Irvine, California 92612
[b]University of California, Irvine, Department of Biomedical Engineering, 3120 Natural Sciences II, Irvine, California 92697-2715
[c]Modulated Imaging Inc., 1002 Health Sciences Road, Irvine California 92612

Abstract. Laser speckle imaging (LSI) is a technique in which coherent light incident on a surface produces a reflected speckle pattern that is related to the underlying movement of optical scatterers, such as red blood cells, indicating blood flow. Image-processing algorithms can be applied to produce speckle flow index (SFI) maps of relative blood flow. We present a novel algorithm that employs the NVIDIA Compute Unified Device Architecture (CUDA) platform to perform laser speckle image processing on the graphics processing unit. Software written in C was integrated with CUDA and integrated into a LabVIEW Virtual Instrument (VI) that is interfaced with a monochrome CCD camera able to acquire high-resolution raw speckle images at nearly 10 fps. With the CUDA code integrated into the LabVIEW VI, the processing and display of SFI images were performed also at ~10 fps. We present three video examples depicting real-time flow imaging during a reactive hyperemia maneuver, with fluid flow through an *in vitro* phantom, and a demonstration of real-time LSI during laser surgery of a port wine stain birthmark. [DOI: 10.1117/1.3528610]

Keywords: speckle interferometry; speckle; real-time imaging; image processing; digital imaging; digital processing.

Paper 10505R received Sep. 15, 2010; revised manuscript received Nov. 12, 2010; accepted for publication Nov. 22, 2010; published online Jan. 28, 2011.

1 Introduction

Laser speckle imaging (LSI) is a noninvasive technique for studying motion of optical scatterers (i.e., red blood cells) with high spatial and temporal resolution. Fercher and Briers[1] first proposed a technique for the analysis of the time-integrated speckle pattern, which results from the interaction of coherent light and a scattering medium. An attractive feature of LSI is its simplicity; the minimum requirements are a laser source, imaging sensor, and a computer for image acquisition and post-processing. Recently, LSI has found uses in monitoring blood flow in the brain,[2] retina,[3] and skin.[4,5]

Conversion of raw laser speckle images to speckle contrast (SC) and speckle flow index (SFI) images is computationally intensive and CPU-based algorithms are not well suited for real-time processing of high-resolution images, let alone real-time visualization of the postprocessed images. One of the reasons for the slow processing times is that the architecture of modern CPUs is optimized for execution of sequential code, with only a fraction of CPU transistors dedicated to arithmetic operations. On the other hand, graphical processing units (GPUs) are well suited for executing mathematical computations on large data sets in parallel, such as rendering graphics for display on a monitor. In addition, multiple GPUs can be added to a system and processing power will scale nearly proportionally.

To exploit the parallel processing power of GPUs for general purpose computing on GPUs, we used NVIDIA's Compute Unified Device Architecture (CUDA). CUDA allows users with programming knowledge of high-level languages, such as C, to take control of the many stream processors of a GPU using the supplied CUDA C extensions.[6] Recently, Liu et al.[7] utilized CUDA to process laser speckle images and were able to obtain a significant speed-up in processing times. However, the algorithms implemented to do so were not clearly defined and a comparison to a fast and efficient CPU algorithm was not addressed. Here, we describe the GPU algorithm in detail and compare the results to an efficient CPU algorithm. In addition, we describe integration of our GPU-based solution with LSI hardware to achieve a complete, real-time blood flow imaging instrument. To enable end users to integrate our GPU-based approach, the CUDA C source code (Appendix A), "Roll" Algorithm C source code (Appendix B), and LabVIEW real-time demonstration software are available for download from our laboratory's website: http://choi.bli.uci.edu.

2 Materials and Methods

In this section, we describe our LSI instrument, the algorithms used to integrate GPU processing into the setup, and visualization of processed images.

2.1 *LSI Setup*

Figure 1 shows the setup for our real-time LSI system. The laser used for our experimental setup was a continuous-wave HeNe laser ($\lambda = 633$ nm, 30 nm, 30 mW; Edmund Industrial Optics, Barrington, New Jersey). Laser light was delivered to

Address all correspondence to: Owen Yang, Beckman Laser Institute and Medical Clinic, University of California, Irvine, 1002 Health Sciences Rd., Irvine, CA 92612. Tel: 949-824-3054; Fax: 949-824-6969; E-mail: yango@uci.edu

1083-3668/2011/16(1)/016009/14/$25.00

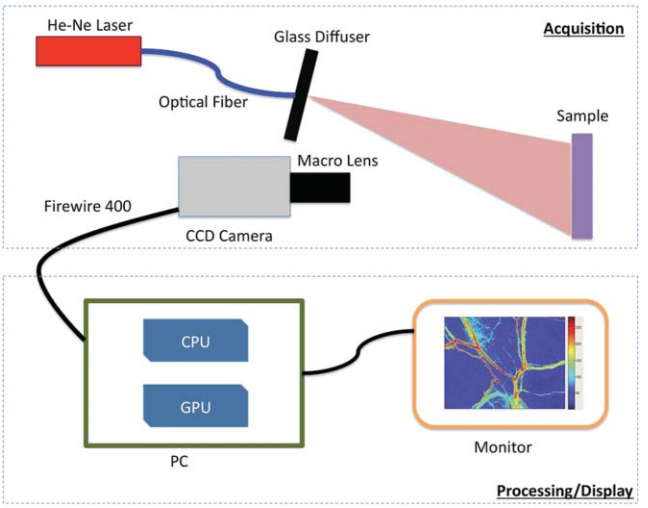


Fig. 1 Schematic of a typical LSI setup consisting of the (a) acquisition stage (He-Ne 633-nm laser, optical fiber, glass diffuser, CCD camera) and (b) processing/display stage (PC and monitor).

the target with an optical fiber and diverged with a ground-glass diffuser (Thorlabs, Newton, New Jersey). Reflected speckle patterns were imaged with a 12-bit, thermoelectrically cooled CCD Camera (RetigaEXi or Retiga 2000R, QImaging, Burnaby, BC, Canada) with a resolution of 1392 (W)×1040 (H) (Retiga EXi) or 1600 (W)×1200 (H) (Retiga 2000R) pixels and a close-focus zoom lens (18–108 mm, f/2.5-close, Edmund Optics, model no. 52–274, Barrington, New Jersey). Images were transferred to the personal computer (PC) in real time via the FireWire 400 interface at a frame rate of ~9.7 fps (RetigaEXi) or 8.3 fps (Retiga 2000R). The PC was running an Intel Core 2 4300 running at 1.80 Ghz, 1 GB DDR2 RAM, and Windows XP Professional with Service Pack 3. To process and render the SC and SFI images, we developed software in Microsoft Visual Studio 2005 and LabVIEW 8.6 (National Instruments, Austin, Texas). The graphics card used for image processing was the GeForce 8800GTS made by EVGA, which included the NVIDIA G80 GPU and 640MB of GDDR3 RAM. The performance specifications included a core clock of 500 MHz, memory clock of 800 MHz, shader clock of 1200 MHz, 12 stream multiprocessors, and 96 stream processors (eight per multiprocessor).

2.2 *Algorithms to Process LSI Data*

To quantify the blurring effect associated with moving particles in raw laser speckle images, the local spatial SC is computed for each pixel using the sliding-window technique (where ω is the radius of the window, usually $\omega = 2$ or $\omega = 3$) and the following:

$$K = \frac{\sigma}{\langle I \rangle} = \frac{\sqrt{[1/(N-1)]\sum_{i=1}^{N}(I_i - \langle I \rangle)^2}}{\langle I \rangle} = \frac{\sqrt{[1/(N-1)]\left(\sum_{i=1}^{N} I_i^2\right) - N\langle I \rangle^2}}{\langle I \rangle}, \quad (1)$$

where σ is the local standard deviation, $\langle I \rangle$ the local mean within the sliding window, N is $(\omega + 1 + \omega)^2$, and I_i is the intensity of pixel i within the sliding window. The last formula contains a simpler form of standard deviation using sum of squares that is used for more efficient processing. For computing edge pixels, the values outside of the actual image are assumed to be zero, resulting in suspect calculations of K at these pixels.

In brief, to relate SC to actual flow rates, the speckle correlation time τ must first be determined. We employed the simplified speckle imaging equation by Ramirez-San-Juan et al.[8] and Cheng and Duong:[9]

$$\tau = 2TK^2, \quad (2)$$

where T is the exposure time. Assuming $1/\tau$ to be proportional to the degree of blood flow, we calculated relative flow values, or SFI values, using the following:

$$\mathrm{SFI} = \frac{1}{\tau} = \frac{1}{2TK^2}. \quad (3)$$

2.3 *LSI Algorithm Implementation in CUDA*

The motivation for using a GPU for this application is its massively parallel architecture, which can be efficiently exploited to compute SC and SFI values much faster than the CPU alone. A raw speckle image is divided into a grid of blocks, called thread blocks, which are processed concurrently by the stream multiprocessors until all blocks are in use. Each multiprocessor has eight stream processors and limited, but extremely fast, on-chip shared memory to be used in the processing of the data in each thread block. To handle the concurrent execution of same command for each thread, NVIDIA uses a new architecture based on single-instruction, multiple-thread that handles the vast number of thread programs in parallel. For this particular application, each thread in the thread block executes identical instructions (called the kernel) for each pixel. Optics-related applications of GPU technology include holography,[10,11] real-time shape measurements,[12] atmospheric turbulence,[13] and Monte Carlo modeling of light transport.[14,15] An excellent description of GPUs is provided in Ref. 16.

Figure 2 depicts a comparison of CPU and GPU methods to convert a single raw speckle image (e.g., 1392×1040) into SC and SFI images. After the main (host) memory has been allocated, the first step for processing raw speckle images on the GPU is to allocate global memory on the graphics card using the cudaMalloc() function. The raw speckle image, SC, and SFI images are both preallocated, along with intermediate temporary matrices. The raw speckle image is then transferred from host memory to GPU memory using the cudaMemcpy() that was preallocated in the previous step. Next, a grid of thread blocks and number of threads per block must be defined, using the dim3 class, prior to execution of the kernel. Once these variables are defined, the kernel(s) (explained in detail later) can be executed. The SC and SFI images are then moved from GPU memory to host memory using the cudaMemcpy() function. Finally, GPU memory is deallocated using the cudaFree() function.

The conversion from raw speckle images to SC and SFI images consists of two separate kernels (C source code is shown

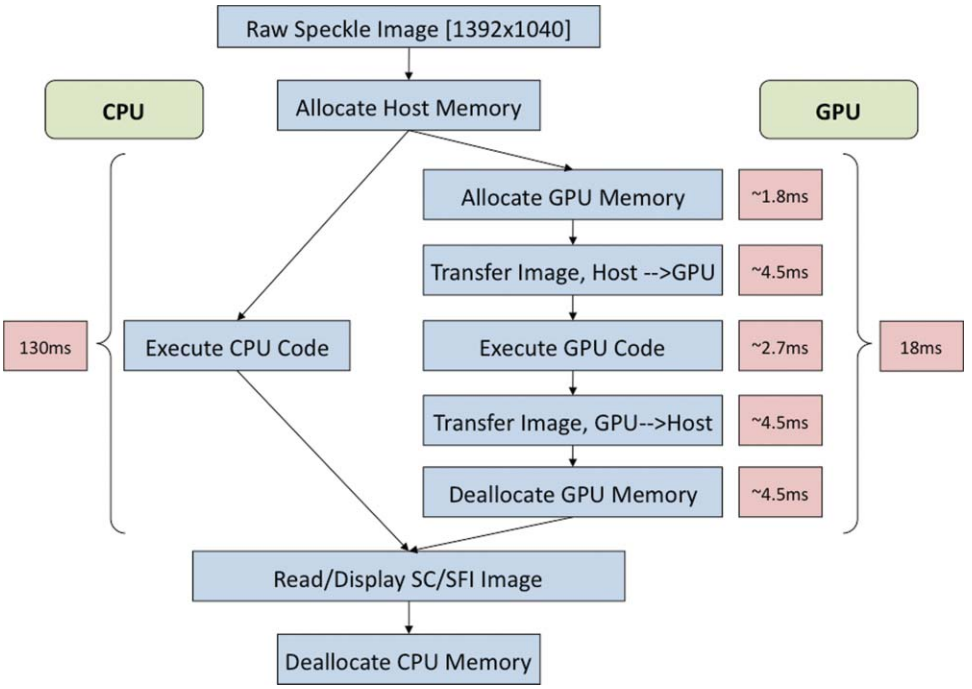


Fig. 2 Flowchart of the differences between utilizing a CPU versus a GPU in converting a single raw image into SC/SFI images. The processing times of individual steps are highlighted in red for an image pixel resolution of 1392×1040. The processing steps and times enclosed in brackets are repeated in real-time software.

in Appendix A), which collectively mimic the separable convolution method for filters and computes the sum of squares for the standard deviation calculations in Eq. (1) using the least amount of computational resources. The first kernel commences by moving data from global GPU memory to the allocated shared memory and then calculates the sum of the row within the sliding window in which the pixel of interest resides (Fig. 3 depicts a $\omega = 2$ sliding-window example). The second kernel similarly initializes shared memory then computes the sum of the aforementioned summed rows to compute squared sums and mean intensity of the sliding window, with minimal numerical calculations (Fig. 4). The second kernel continues by computing the

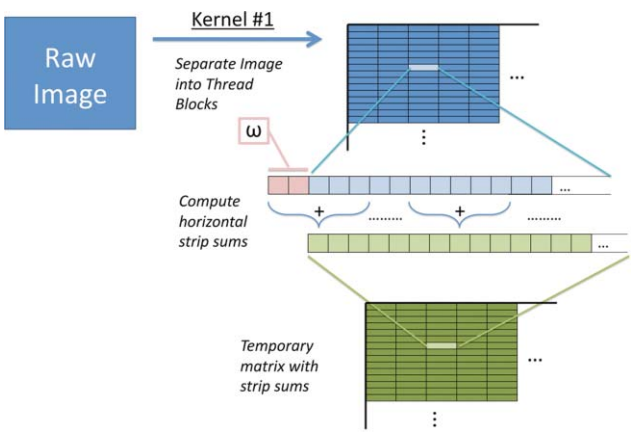


Fig. 3 First of two kernels executed on the GPU to convert raw images into SC/SFI images. A raw image is separated into smaller thread blocks to be executed by the multiprocessors on the GPU. The individual threads compute the sum of a $(\omega + 1 + \omega)$ width strip in preparation for mean and standard deviation calculations [see Eq. (1)].

sample standard deviation in the numerator of Eq. (1). After the SC has been computed for each pixel, a quick calculation using Eq. (3) results in an SFI map.

2.4 *Real-Time LabVIEW LSI Visualization with GPU Integration*

Because of the relative ease of interfacing our QImaging camera with the computer and displaying both raw and processed images, LabVIEW was employed as the visualization platform for our GPU code. To integrate the LSI CUDA C code into LabVIEW, three C wrapper functions are written: LSIonGPU_init(), to initialize the memory on the graphics card, LSIonGPU_process(), to transfer the raw speckle image into GPU space, execute both kernels, and transfer the SC and SFI images back to the host, and lastly LSIonGPU_term() to deallocate GPU memory so that it can be used for other purposes. The entire C project is compiled as a dynamic link library (DLL) so that the functions can be called from LabVIEW.

For maximum performance, LabVIEW code is written with two main while loops, one for image acquisition from the attached camera and another for image processing, which are executed simultaneously (Fig. 5). The purpose for this approach is to enable simultaneous acquisition and processing of images (executed in parallel on a dual-core machine), thus maximizing the frame rate of the system. For this approach to perform properly, a ring buffer must be employed. A ring buffer allocates sufficient memory such that multiple raw images can fit within it. As a new image is acquired, it fills the next space within the buffer and when it reaches the end, it loops around and replaces the first slot, then second, etc. This method aids in ensuring that the processing while loop has enough time to process the

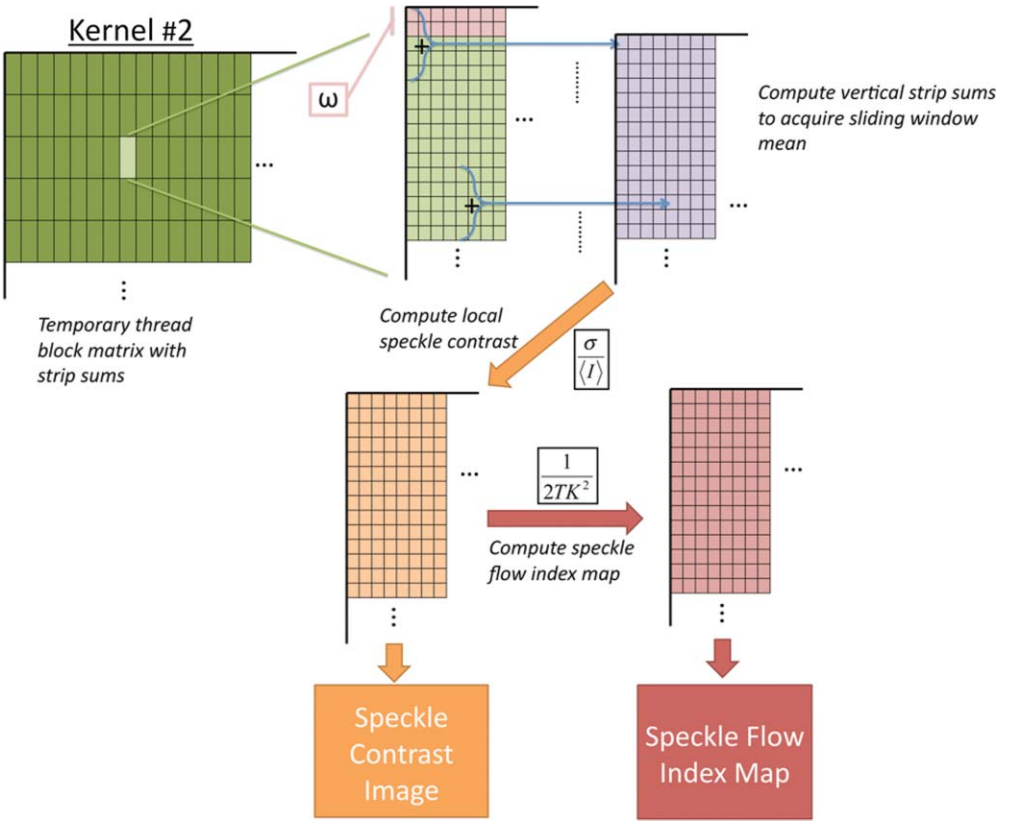


Fig. 4 Second of two kernels executed on the GPU to convert raw images into SC/SFI images. The strip sums calculated in Kernel 1 are summed in the vertical direction and used to calculate the mean and standard deviation of the sliding window for every pixel in the image. The SC and SFI images are subsequently computed.

data before being replaced by freshly acquired images from the camera.

Another important feature in the design of this program is the use of the queue buffer feature in LabVIEW. A queue buffer allows the transfer of data between different parts of a block diagram—in this case, the two while loops. As soon as an image is acquired from the camera and placed into the ring buffer, an element is enqueued into the queue buffer with a reference to the location in which the image was placed. As soon as an element is within the queue buffer, the processing while loop will dequeue the foremost element and can immediately begin processing the dequeued data. These two features in combination allow immediate processing of raw speckle images without the downtime of waiting for processing to finish before a new image is acquired from the camera. As long as the processing time is shorter than the image acquisition time, the proposed algorithm can execute indefinitely.

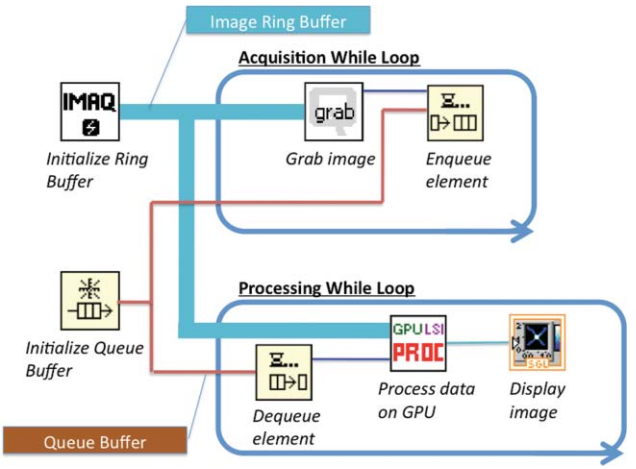


Fig. 5 LabVIEW pseudocode used to acquire and process data from the camera. Two for loops are employed such that images can be processed while newer images are acquired. An image ring buffer is used to store multiple images in memory and the queue buffer is used to pass information between the two for loops.

3 Results

This section discusses the results of integrating the GPU into the laser speckle image processing model. As a proof of concept, video demonstrations utilizing the custom LabVIEW program are shown.

3.1 *LSI on CPU versus GPU C Performance Benchmarking*

To make a proper comparison between the performances of a CPU versus a GPU, an efficient CPU algorithm was implemented beforehand. Tom et al.[17] devised an algorithm named the "Roll" algorithm to convert raw speckle images into SC images. The algorithm takes the rolling sums of rows and columns and subtracts the new row/column to minimize the number of

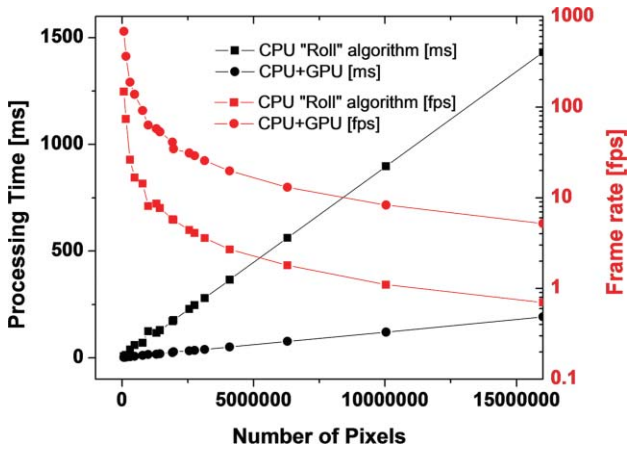


Fig. 6 Graph of processing times for various image sizes (i.e., number of pixels).

calculations needed to compute the standard deviation of the sliding window. We integrated their algorithm in customized C code and performed benchmarking experiments (C source code is shown in Appendix B).

The timer used to measure system performance for both CPU and GPU algorithms was the built-in timer function in CUDA. Both the CPU and GPU method used a 5×5 sliding-window filter and was performed on randomly generated images generated in C using the rand() function. The results for processing a single image at various image resolutions are displayed in Fig. 6 and compiled in Table 1. For CPU algorithm benchmarking, we exclude the time it takes to allocate free memory due to the fact that the our LabVIEW code only allocates/deallocates memory one time after the program has started, thus making the associated memory management time irrelevant in real-time execution of our code. Similarly, the GPU benchmarking values exclude the time it takes to allocate and deallocate memory on the GPU in addition to the host memory deallocation/allocation times due to the fact that these processes are only performed once during run time. It is important to note that the GPU algorithm times only include the time it takes to transfer the raw image onto the GPU, execute both kernels, and transfer the processed image back to main memory. The normalized errors [(LSI_CPU – LSI_GPU)/(LSI_CPU)] were computed for all pairs of images and found to be negligible, with errors on the order of 10^{-7}.

3.2 *Real-Time LabVIEW Performance*

With the setup described in Fig. 1, the LabVIEW software is evaluated with a single raw image (1392×1040) being processed with the GPU and visualized with the built-in Intensity Graph. The maximum frame rate is ~15 fps, which is sufficient to display the real-time LSI images because the maximum frame rate of our camera is ~10 fps (RetigaEXi). As such, the program can run indefinitely without memory conflicts because the processing and display times are less than the time between collection of successive raw speckle images. In addition, as GPUs gain fast clock speeds, more processing cores, and faster RAM, the gap will widen further.

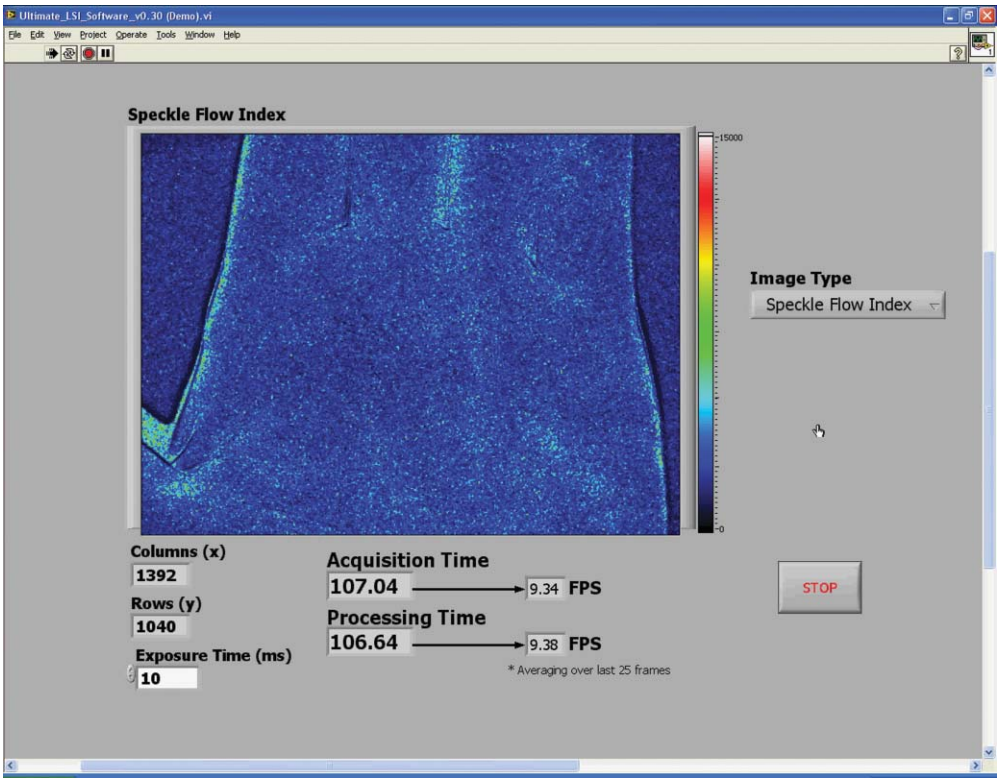
Table 1 CPU versus GPU processing times for various image sizes.

LSI processing times for various image sizes (5×5 window)				
Resolution (W×H)	Pixels	CPU (ms)	GPU (ms)	Speed-up
320×240	76,800	6.8	1.5	4.6×
480×320	153,600	13.5	2.8	4.9×
640×480	307,200	38.0	5.3	7.2×
800×600	480,000	59.8	7.2	8.3×
1024×768	786,432	69.8	10.9	6.4×
1000×1000	1,000,000	124.1	15.8	7.9×
1280×1024	1,310,720	116.7	17.3	6.7×
1200×1200	1,440,000	129.2	18.7	6.9×
1392×1040	1,447,680	130.2	18.8	6.9×
1600×1200	1,920,000	171.8	24.4	7.1×
1400×1400	1,960,000	176.4	28.8	6.1×
1600×1600	2,560,000	228.4	32.1	7.1×
1920×1440	2,764,800	246.8	34.4	7.2×
2048×1536	3,145,728	280.3	39.0	7.2×
1560×1600	4,096,000	365.5	50.5	7.2×
3072×2048	6,291,456	562.2	76.9	7.3×
3872×2592	10,036,224	897.6	120.4	7.5×
4000×4000	16,000,000	1431.7	192.1	7.5×

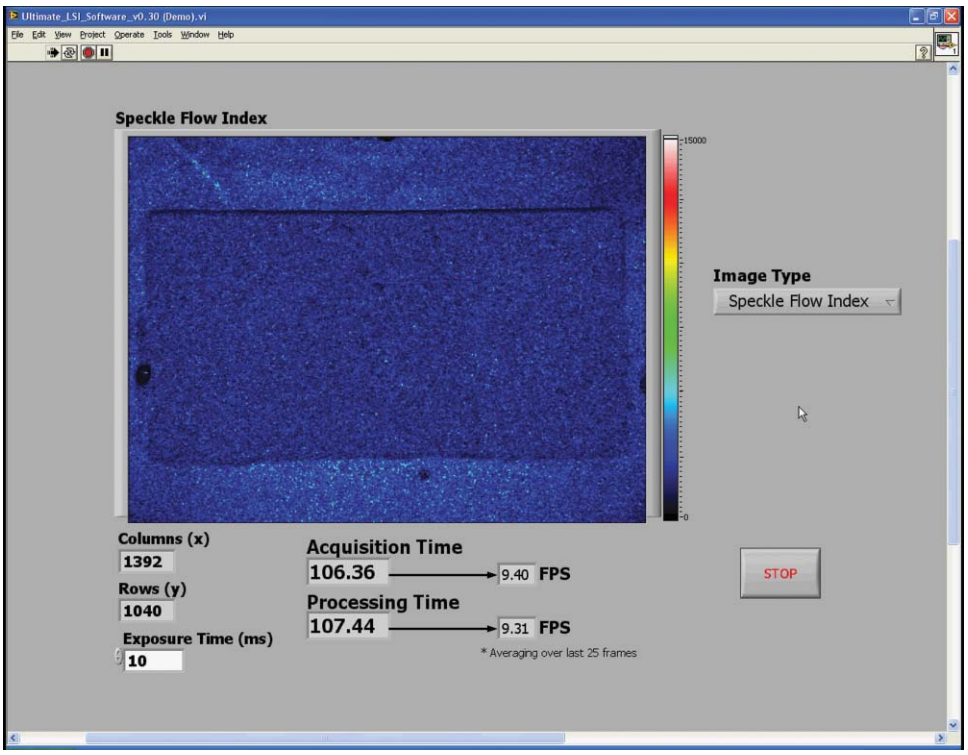
3.2.1 *Real-time LabVIEW video demonstration 1—reactive hyperemia*

This section contains videos of the fully functional LabVIEW software described in Section 2.3. The videos are acquired with a simple screen capture utility that captures the entire LabVIEW window frame. Videos are taken with ambient lighting off and exposure time set to 10 ms. Please note that the video-capture software in conjunction with a relatively high overhead program, such as LabVIEW, slightly reduces the image-processing speed of the system by ~1 fps.

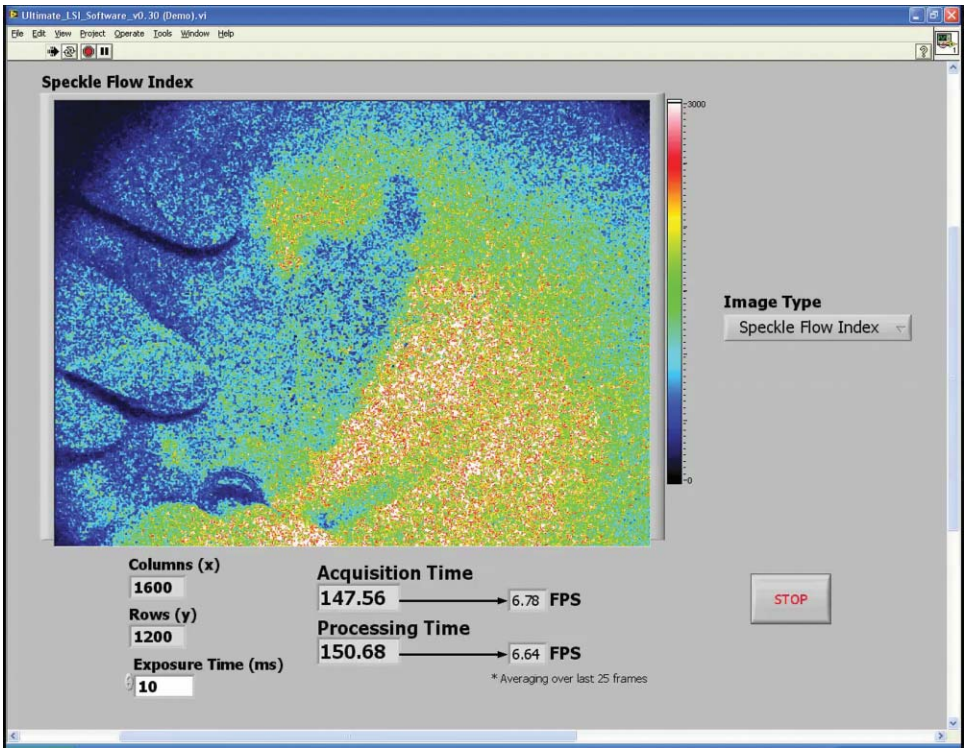
The first demonstration consists of viewing real-time SFI images displayed in LabVIEW's built-in Intensity Graph of a reactive hyperemia experiment of theinner hand (Video 1). A pressure cuff was placed over the upper arm of a subject (IRB Protocol no. 2004-3626). The palm side of the subject's hand was imaged for 3 min without any applied pressure. The cuff was inflated to 220 mm Hg for 3 min, after which the applied pressure was abruptly released. Video 1 shows real-time SFI images collected during the ~20 s before the pressure cuff was released and the ensuing ~20 s after release. Immediately after



Video 1 Single-frame excerpt from LabVIEW real-time LSI demonstration of reactive hyperemia experiment (MPEG, 3 MB) [URL: http://dx.doi.org/10.1117/1.3528610.1]



Video 2 Single-frame excerpt from LabVIEW real-time LSI demonstration of ordered fluid flow (MPEG, 3 MB). [URL: http://dx.doi.org/10.1117/1.3528610.2]



Video 3 Single-frame excerpt from LabVIEW real-time LSI demonstration of PWS therapy (MPEG, 7 MB).
[URL: http://dx.doi.org/10.1117/1.3528610.3]

release of the pressure cuff, a typical hyperemic response was observed, with a large-scale influx of blood into the hand.

3.2.2 *Real-time LabVIEW video demonstration 2—ordered flow in phantom*

The second real-time demonstration involves use of the same instrument and software to image, in real-time, ordered fluid flow (Video 2). A simple ordered-flow phantom was created by attaching plastic tubing to a piece of cardboard. The video depicts manually controlled injection of a scattering fluid (20% Intralipid) into the tube with a syringe, followed by manual retraction of the fluid halfway through the video. The position of the leading edge of the fluid moving through the tube can be visualized with high temporal resolution, limited in this case only by the maximum frame rate (~10 fps) of the camera.

3.2.3 *Real-time LabVIEW video demonstration 3—intraoperative LSI during port wine stain therapy*

Port wine stain (PWS) birthmarks are vascular malformations characterized by ectatic blood vessels in the dermis of the skin.[18] PWS therapy involves the use of a laser to photocoagulate the blood vessels using a wavelength with high oxy-hemoglobin absorption [e.g., pulsed dye laser (PDL), 577 nm].[19] The LSI system (using the Retiga 2000R camera) was brought into the operating room of the Surgery Laser Clinic adjoining Beckman Laser Institute, and screen-capture video was recorded before, during, and after PDL treatment of PWS. All imaging procedures were approved by the Institutional Review Board at University of California, Irvine (Video 3) depicts the changes in SFI values in the skin of treated areas. The hand of the surgeon and PDL handpiece are intermittently shown within the video footage. The reduction in relative blood flow is immediately apparent and visible for the surgeon to make an assessment of the degree of photocoagulation of PWS vessels.

4 Conclusions

We have demonstrated a full implementation of real-time visualization of blood flow, using the GPU as the main processing unit and integrated the technology into a LabVIEW-based program. The parallelized architecture of the GPU is able to process data faster than the acquisition rate of our camera and results in our system being limited by the bus speed of the camera. Thus, parallel acquisition and processing of data using the GPU allows for an efficient method for real-time visualization of laser speckle images.

Although the video demos presented here have a slight decrease in visualized frame per second, this is mainly due to the processing overhead associated with LabVIEW. The algorithm to process data is certainly capable of higher frame rates. This system can also be modified with a higher-frame-rate camera while still maintaining very high frame rates. For example, if a 1920 (W)×1440 (H) camera was used, the GPU-based system can process data at ~30 fps, whereas a CPU-only–based system will result in ~4 fps. In addition, the flexibility of CUDA enables increased processing speed simply by integrating additional GPUs into the system.

We have also demonstrated the effectiveness of our system by monitoring real-time LSI processed images (Video 3). The relative changes in motion on a macroscopic scale can be visual-

ized for both bulk perfusion and ordered fluid flow. Additionally, preliminary measurements with our real-time LSI system during PWS treatment clearly show a reduction in blood flow, as one would expect with PDL therapy. Therefore, we envision immediate biomedical applications of such an instrument toward image-guided surgery and physiological studies.

Appendix A

```
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Kernel Code that is executed on GPU to convert Raw to SC/SFI
// Two kernels (rows and columns) execute sequentially
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

// 24-bit multiplication is faster on G80,
// but we must be sure to multiply integers
// only within [-8M, 8M - 1] range
#define IMUL(a, b) __mul24(a, b)

#ifndef _SPECKLECONTRAST_H_
#define _SPECKLECONTRAST_H_

////////////////////////////////////////////////////////////////////////////////////////
// Kernel configuration (constants for GPU calculations)
////////////////////////////////////////////////////////////////////////////////////////
// Assuming ROW_TILE_W, KERNEL_RADIUS_ALIGNED and dataW
// are multiples of maximum coalescable read/write size,
// all global memory operations are coalesced in convolutionRowGPU5()
#define         ROW_TILE_W 128
#define WINDOW_RADIUS_ALIGNED 16

// Assuming COLUMN_TILE_W and dataW are multiples
// of maximum coalescable read/write size, all global memory operations
// are coalesced in convolutionColumnGPU5()
#define COLUMN_TILE_W 16
#define COLUMN_TILE_H 48

////////////////////////////////////////////////////////////////////////////////////////
// Loop unrolling templates, needed for best performance (5×5 window)
////////////////////////////////////////////////////////////////////////////////////////
template<int i> __device__ int convolutionRowMeans5(int *data){
    return data[2 - i] + convolutionRowMeans5<i - 1>(data);
}

template<> __device__ int convolutionRowMeans5<-1>(int *data){
    return 0;
}

template<int i> __device__ int convolutionRowSquares5(int *data){
    return data[2 - i] * data[2 - i] + convolutionRowSquares5<i - 1>(data);
}

template<> __device__ int convolutionRowSquares5<-1>(int *data){
    return 0;
}
template<int i> __device__ int convolutionColumnFull5(int *data){
    return data[(2 - i) * COLUMN_TILE_W] + convolutionColumnFull5<i - 1>(data);
}

template<> __device__ int convolutionColumnFull5<-1>(int *data){
    return 0;
}
```

```
////////////////////////////////////////////////////////////////////////////////
// Row convolution filter (5×5)
////////////////////////////////////////////////////////////////////////////////
// dataW and dataH are the width and height of raw data, respectively
// d_Data, d_Means, d_Squares are preallocated device memory for the raw input,
//    strip sums of mean buffer, and strip sums of squares buffer, respectively
// WINDOW_RADIUS is NOT current implemented, 5×5 window filter is hardcoded
__global__ void convolutionRowGPU5(
   int *d_Data,
   int *d_Means,
        int *d_Squares,
   int dataW,
   int dataH,
     int WINDOW_RADIUS
){
   // Shared memory allocation
   __shared__ int data[2 + ROW_TILE_W + 2];
   // Current tile and apron limits, relative to row start
   const int      tileStart = IMUL(blockIdx.x, ROW_TILE_W);
   const int        tileEnd = tileStart + ROW_TILE_W - 1;
   const int     apronStart = tileStart - 2;
   const int       apronEnd = tileEnd + 2;

   // Clamp tile and apron limits by image borders
   const int        tileEndClamped = min(tileEnd, dataW - 1);
   const int    apronStartClamped = max(apronStart, 0);
   const int      apronEndClamped = min(apronEnd, dataW - 1);

   // Row start index in d_Data[]
   const int       rowStart = IMUL(blockIdx.y, dataW);

   // Aligned apron start. Assuming dataW and ROW_TILE_W are multiples
   // of half-warp size, rowStart + apronStartAligned is also a
   // multiple of half-warp size, thus having proper alignment
   // for coalesced d_Data[] read.
   const int   apronStartAligned = tileStart - WINDOW_RADIUS_ALIGNED;
   // Current load postion (of this thread)
   const int loadPos = apronStartAligned + threadIdx.x;
   // Set the entire data cache contents
   // Load global memory values, if indices are within the image borders,
   // or initialize with zeroes otherwise
   if(loadPos > = apronStart){
        //shared memory postion
        const int smemPos = loadPos - apronStart;

        data[smemPos] =
             ((loadPos > = apronStartClamped) && (loadPos < = apronEndClamped)) ?
             d_Data[rowStart + loadPos] : 0;
        }
   // Ensure the completeness of the loading stage
   // because results, emitted by each thread depend on the data,
   // loaded by another threads
   __syncthreads();

   // Current write position
   const int writePos = tileStart + threadIdx.x;
   // Assuming dataW and ROW_TILE_W are multiples of half-warp size,
   // rowStart + tileStart is also a multiple of half-warp size,
   // thus having proper alignment for coalesced d_Result[] write.
```

```
   if(writePos < = tileEndClamped){
        const int smemPos = writePos - apronStart;
        int sumMeans   = 0;
        int sumSquares = 0;

// Loop unrolling section - computes rolling sums and square sums
// only uses unrolling templates (defined at begining) if defined
// if not defined, uses for loop
#ifdef UNROLL_INNER
        sumMeans   = convolutionRowMeans5<2 * 2>(data + smemPos);
        sumSquares = convolutionRowSquares5<2 * 2>(data + smemPos);
#else
        for(int k = -2; k < = 2; k+ +){
                  sumMeans   + = data[smemPos + k];
                  sumSquares + = data[smemPos + k] * data[smemPos + k];
                  }
#endif
                    // Writes the strip sums to preallocated global memory
          d_Means[rowStart + writePos] = sumMeans;
                    d_Squares[rowStart + writePos] = sumSquares;
     }
}
////////////////////////////////////////////////////////////////////////////////
// Column convolution filter + Speckle Contrast + Speckle Flow Index (5×5)
////////////////////////////////////////////////////////////////////////////////
// dataW and dataH are the width and height of raw data, respectively
// d_MeansRow, d_SquaresRow are preallocated device memory for strip sums of
//       mean and squares values obtained in the previous kernel, respectively
// d_MeansFull, d_SquaresFull are preallocated device memory for full sums of
//       mean and squares values buffer, respectively
// d_SC, d_SFI are preallocated device memory for speckle contrast and speckle
//       flow index values buffer, respectively
// expT is the exposure time of the raw images
// smemStride, gmemStride are the incremental position movement of the
// shared and globably memory processing positions, respectively
// WINDOW_RADIUS is NOT current implemented, 5×5 window filter is hardcoded
__global__ void convolutionColumnGPU5(
   int *d_MeansRow,
   int *d_SquaresRow,
   int *d_MeansFull,
   int *d_SquaresFull,
   float *d_SC,
   float *d_SFI,
   int dataW,
   int dataH,
   int WINDOW_RADIUS,
   float expT,
   int smemStride,
   int gmemStride
){
   // Shared memory allocation
   __shared__ int rowMeans[COLUMN_TILE_W * (2 + COLUMN_TILE_H + 2)];
   __shared__ int rowSquares[COLUMN_TILE_W * (2 + COLUMN_TILE_H + 2)];
   // Current tile and apron limits, in rows
   const int          tileStart = IMUL(blockIdx.y, COLUMN_TILE_H);
   const int            tileEnd = tileStart + COLUMN_TILE_H - 1;
   const int         apronStart = tileStart - 2;
   const int           apronEnd = tileEnd + 2;
   // Clamp tile and apron limits by image borders
   const int     tileEndClamped = min(tileEnd, dataH - 1);
```

```
   const int  apronStartClamped = max(apronStart, 0);
   const int    apronEndClamped = min(apronEnd, dataH - 1);

   // Current column index
   const int         columnStart = IMUL(blockIdx.x, COLUMN_TILE_W) + threadIdx.x;
   const int num_WINDOW_ELEMENTS = IMUL(WINDOW_RADIUS+1+WINDOW_RADIUS,
WINDOW_RADIUS+1+WINDOW_RADIUS);

   // Shared and global memory indices for current column
   int smemPos = IMUL(threadIdx.y, COLUMN_TILE_W) + threadIdx.x;
   int gmemPos = IMUL(apronStart + threadIdx.y, dataW) + columnStart;
   // Cycle through the entire data cache
   // Load global memory values, if indices are within the image borders,
   // or initialize with zero otherwise
   for(int y = apronStart + threadIdx.y; y < = apronEnd; y + = blockDim.y){
        rowMeans[smemPos] =
        ((y > = apronStartClamped) (y < =   apronEndClamped)) ?
        d_MeansRow[gmemPos] : 0;
        rowSquares[smemPos] =
        ((y > = apronStartClamped) (y < =   apronEndClamped)) ?
        d_SquaresRow[gmemPos] : 0;
        smemPos + = smemStride;
        gmemPos + = gmemStride;
   }
   // Ensure the completeness of the loading stage
   // because results, emitted by each thread depend on the data,
   // loaded by another threads
   __syncthreads();

   // Shared and global memory indices for current column
   smemPos = IMUL(threadIdx.y + WINDOW_RADIUS, COLUMN_TILE_W) + threadIdx.x;
   gmemPos = IMUL(tileStart + threadIdx.y, dataW) + columnStart;
   // Cycle through the tile body, clamped by image borders
   // Calculate the sum of the strips sums (and squares)
   for(int y = tileStart + threadIdx.y; y < = tileEndClamped; y + = blockDim.y){
        int sumMeansFull = 0;
        int sumSquaresFull = 0;

// Loop unrolling section
// only uses unrolling templates (defined at beginning) if defined
// if not defined, uses for loop
#ifdef UNROLL_INNER
        sumMeansFull = convolutionColumnFull5<2 *   2>(rowMeans + smemPos);
        sumSquaresFull = convolutionColumnFull5<2 * 2>   (rowSquares + smemPos);
#else
     for(int k = -2; k < = 2; k+ +){
            sumMeansFull + =
             rowMeans[smemPos + IMUL(k, COLUMN_TILE_W)];
            sumSquaresFull + =
             rowSquares[smemPos + IMUL(k, COLUMN_  TILE_W)];
     }
#endif
            // Write sums to device memory
            d_MeansFull[gmemPos]   = sumMeansFull;
            d_SquaresFull[gmemPos] = sumSquaresFull;

            // Calcuation of Speckle Contrast (K)
            // K = standard deviation / mean
            // Compute mean
            float mean = sumMeansFull/(float)num_WINDOW_ELEMENTS;
```

```
            // Compute standard deviation
            float coeff = __fdividef(1,num_WINDOW_ELEMENTS-1);
            float SD = sqrtf(coeff*(sumSquaresFull -
            (num_WINDOW_ELEMENTS*mean*mean)));
            float K = SD/mean;

            // Calculation of Speckle Flow Index (SFI)
            // SFI = 1 / (2 * tc * K * K)
            // tc = correlation time = 10ms

            float SFI = 1 / (2 * expT * K * K);

            // move calculated values into master Speckle Contrast matrix
            d_SC[gmemPos] = K;

            // move calculated values into master Speckle Flow matrix

            d_SFI[gmemPos] = SFI;
      smemPos + = smemStride;
      gmemPos + = gmemStride;
   }
}
```

Appendix B

```
////////////////////////////////////////////////////////////////////////////////
// 'Roll' Algorithm, Custom Integration based off Tom et al. (Ref. 17)
////////////////////////////////////////////////////////////////////////////////
// width and height are the width and height of the raw image, respecively
// Raw is the raw speckle image
// SpeckleContrast and SpeckleFlowIndex are preallocated 1D matricies
// rollRow, rollColumn, rollRowSquares, rollColumnSquared are preallocated
// buffers for holding the strip sums in the row and column and strip sums
// of squared values in the row and column, respectively
// the size of these 4 buffers are width*1
// wR is the size of the sliding window radius
// t is the exposure time of the camera used to obtain the raw speckle images
void LSI_roll(int *Raw, float *SpeckleContrast, float *SpeckleFlowIndex,
            int *rollRow, int *rollColumn,
            int *rollRowSquared, int *rollColumnSquared,
            int width, int height,
            int wR, float t)
{
// Full window size
int w = wR*2 + 1;
// Number of elements within sliding window
int els = w*w;
// Inverse degrees of freedom
float iDoF = (float)1.0/(els*(els-1));
// Intialize counters
int i, j;
// Step 1) Calculate first accumulated sum to start
// Perform analysis on first accumulated row
for (i = 0; i<width; i++)
{
for (j = 0; j<w; j++)
{
rollRow[i] + = Raw[j*width+i];
rollRowSquared[i] + = (Raw[j*width+i]*Raw[j*width+i]);
}
```

```
if (i> = w)
{
rollColumn[i] = rollColumn[i-1] - rollRow[i-w] + rollRow[i];
rollColumnSquared[i] = rollColumnSquared[i-1] - rollRowSquared[i-w] +
     rollRowSquared[i];
SpeckleContrast[wR*width+i-wR] = (els * sqrt((float)iDoF*(els*
     rollColumnSquared[i]-(rollColumn[i]*rollColumn[i])))/rollColumn[i];
SpeckleFlowIndex[wR*width+i-wR] = (float)1/(2*SpeckleContrast[wR*width+i-wR]*
     SpeckleContrast[wR*width+i-wR]*t);
}
else
{
rollColumn[w-1] + = rollRow[i];
rollColumnSquared[w-1] + = rollRowSquared[i];
if(i = = (w-1))
{
SpeckleContrast[wR*width+i-wR] = (els * sqrt((float)iDoF* (els*
     rollColumnSquared[w-1]-(rollColumn[w-1]*rollColumn[w-1])))/rollColumn[w-1];
SpeckleFlowIndex[wR*width+i-wR] = (float)1/(2*SpeckleContrast[wR*width+iwR]*
     SpeckleContrast[wR*width+i-wR]*t);
}
}
}
// Step 2) Perform optimized roll algorithm
// Cumulate rows and columns simultaneously
// Calculate SC and SFI immediately
for (j = w; j<height; j+ +)
{
rollColumn[w-1] = 0;
rollColumnSquared[w-1] = 0;
for (i = 0; i<width; i+ +)
{
rollRow[i] = rollRow[i] - Raw[(j-w)*width+i] + Raw[j*width+i];
rollRowSquared[i] = rollRowSquared[i] - (Raw[(j-w)*width+i]*Raw[(j-w)*width+
     i]) + (Raw[j*width+i]*Raw[j*width+i]);
if (i> = w)
{
rollColumn[i] = rollColumn[i-1] - rollRow[i-w] + rollRow[i];
rollColumnSquared[i] = rollColumnSquared[i-1] --
     rollRowSquared[i-w] + rollRowSquared[i];
SpeckleContrast[(j-wR)*width+i-wR] = (els * sqrt((float)iDoF*(els*
     rollColumnSquared[i]-(rollColumn[i]* rollColumn[i])))/rollColumn[i];
SpeckleFlowIndex[(j-wR)*width+i-wR] = (float)1/(2*SpeckleContrast[(j-wR)*
width+i-wR]*SpeckleContrast[(j-wR)*width+i-wR]*t);
}
else
{
rollColumn[w-1] + = rollRow[i];
rollColumnSquared[w-1] + = rollRowSquared[i];
if(i = = (w-1))
{
SpeckleContrast[(j-wR)*width+i-wR] = (els * sqrt((float)iDoF*(els*
     rollColumnSquared[w-1]-(rollColumn[w-1]*rollColumn[w-1])))/rollColumn[w-1];
SpeckleFlowIndex[(j-wR)*width+i-wR] = (float)1/(2*SpeckleContrast[(jwR)*
     width+i-wR]*SpeckleContrast[(j-wR)*width+i-wR]*t);
}
}
}
}
}
```

Acknowledgments

This work was funded, in part, by the Arnold and Mabel Beckman Foundation, the National Institutes of Health (Grant No. EB0095571, to B.C.), and the National Institutes of Health (NIH) Laser Microbeam and Medical Program (Grant No. P41-RR01192). The authors acknowledge the contributions of Austin McElroy and also the contributions of Dr. J. Stuart Nelson for his permission to perform LSI during laser surgery. The authors acknowledge Bruce Yang and Eugene Huang for technical support on the instrumentation and assistance with imaging in the operating room.

References

1. A. F. Fercher and J. D. Briers, "Flow visualization by means of single-exposure speckle photography," *Opt. Commun.* **37**(5), 326–330 (1981).
2. A. K. Dunn, H. Bolay, M. A. Moskowitz, and D. A. Boas, "Dynamic imaging of cerebral blood flow using laser speckle," *J. Cereb. Blood Flow Metab.* **21**(3), 195–201 (2001).
3. H. Cheng, Y. Yan, and T. Q. Duong, "Temporal statistical analysis of laser speckle images and its application to retinal blood-flow imaging," *Opt. Express* **16**(14), 10214–10219 (2008).
4. Y. C. Huang, N. Tran, P. R. Shumaker, K. Kelly, E. V. Ross, J. S. Nelson, and B. Choi, "Blood flow dynamics after laser therapy of port wine stain birthmarks," *Lasers Surg. Med.* **41**(8), 563–571 (2009).
5. B. Choi, W. Jia, J. Channual, K. M. Kelly, and J. Lotfi, "The importance of long-term monitoring to evaluate the microvascular response to light-based therapies," *J. Invest. Dermatol.* **128**(2), 485–488 (2008).
6. NVIDIA CUDA Programming Guide, v.2.3, NVIDIA, Santa Clara, CA (2009).
7. S. Liu, P. Li, and Q. Luo, "Fast blood flow visualization of high-resolution laser speckle imaging data using graphics processing unit," *Opt. Express* **16**(19), 14321–14329 (2008).
8. J. C. Ramirez-San-Juan, R. Ramos-Garcia, I. Guizar-Iturbide, G. Martinez-Niconoff, and B. Choi, "Impact of velocity distribution assumption on simplified laser speckle imaging equation," *Opt. Express* **16**(5), 3197–3203 (2008).
9. H. Cheng and T. Q. Duong, "Simplified laser-speckle-imaging analysis method and its application to retinal blood flow imaging," *Opt. Lett.* **32**(15), 2188–2190 (2007).
10. T. Shimobaba, Y. Sato, J. Miura, M. Takenouchi, and T. Ito, "Real-time digital holographic microscopy using the graphic processing unit," *Opt. Express* **16**(16), 11776–11781 (2008).
11. N. Masuda, T. Ito, T. Tanaka, A. Shiraki, and T. Sugie, "Computer generated holography using a graphics processing unit," *Opt. Express* **14**(2), 603–608 (2006).
12. S. Zhang and S. T. Yau, "High-resolution, real-time 3D absolute coordinate measurement based on a phase-shifting method," *Opt. Express* **14**(7), 2644–2649 (2006).
13. L. Hu, L. Xuan, D. Li, Z. Cao, Q. Mu, Y. Liu, Z. Peng, and X. Lu, "Real-time liquid-crystal atmosphere turbulence simulator with graphic processing unit," *Opt. Express* **17**(9), 7259–7268 (2009).
14. E. Alerstam, T. Svensson, and S. Andersson-Engels, "Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration," *J. Biomed. Opt.* **13**(6), 060504 (2008).
15. Q. Fang and D. A. Boas, "Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units," *Opt. Express* **17**(22), 20178–20190 (2009).
16. J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *J. Comput. Chem.* **28**(16), 2618–2640 (2007).
17. W. J. Tom, A. Ponticorvo, and A. K. Dunn, "Efficient processing of laser speckle contrast images," *IEEE Trans. Med. Imaging* **27**(12), 1728–1738 (2008).
18. B. Tallman, O. T. Tan, J. G. Morelli, J. Piepenbrink, T. J. Stafford, S. Trainor, and W. L. Weston, "Location of port-wine stains and the likelihood of ophthalmic and/or central nervous system complications," *Pediatrics* **87**(3), 323–327 (1991).
19. J. S. Nelson and J. Applebaum, "Clinical management of port-wine stain in infants and young children using the flashlamp-pulsed dye laser," *Clin. Pediatr. (Phila.)* **29**(9), 503–508; discussion 509 (1990).