GPU-Based Real-Time Small Displacement Estimation With Ultrasound

Stephen Rosenzweig, Mark Palmeri, *Member, IEEE*, and Kathryn Nightingale, *Member, IEEE*

***Abstract*—General purpose computing on graphics processing units (GPUs) has been previously shown to speed up computationally intensive data processing and image reconstruction algorithms for computed tomography (CT), magnetic resonance (MR), and ultrasound images. Although some algorithms in ultrasound have been converted to GPU processing, many investigative ultrasound research systems still use serial processing on a single CPU. One such ultrasound modality is acoustic radiation force impulse (ARFI) imaging, which investigates the mechanical properties of soft tissue. Traditionally, the raw data are processed offline to estimate the displacement of the tissue after the application of radiation force. It is highly advantageous to process the data in real-time to assess their quality and make modifications during a study. In this paper, we present algorithms for efficient GPU parallel processing of two widely used tools in ultrasound: cubic spline interpolation and Loupas' two-dimensional autocorrelator for displacement estimation. It is shown that a commercially available graphics card can be used for these computations, achieving speed increases up to 40× compared with single CPU processing. Thus, we conclude that the GPU-based data processing approach facilitates real-time (i.e., <1 second) display of ARFI data and is a promising approach for ultrasonic research systems.**

I. Introduction

There has recently been much investigation into using graphics processing units (GPUs) instead of CPUs for scientific computing. To facilitate using graphics cards for scientific computing, NVIDIA (Santa Clara, CA) introduced Compute Unified Device Architecture (CUDA) to allow users direct access to program the GPUs on NVIDIA graphics cards [1]. The graphics cards' architecture contains a large number of processing cores (64 to 512), each with a small amount of local memory (2 kB), all of which can be used in parallel. To utilize the full power of the GPUs, it is necessary to redesign algorithms to work in a parallel environment rather than a conventional serial environment. It has been shown that GPUs can significantly increase the speed of computed tomography (CT) image reconstruction [2]–[5], magnetic resonance (MR) image reconstruction [6], [7], multi-modality image registration [8], ultrasound scan conversion [9], and ultrasound Doppler flow processing [10].

Ultrasound researchers could clearly benefit from GPU-based processing of raw data to achieve real-time displays for their investigation methods. One such example is acoustic radiation force impulse (ARFI) imaging, which utilizes ultrasound to visualize the mechanical properties of tissue [11], [12]. High-intensity focused acoustic beams are used to generate acoustic radiation force within the tissue, causing small displacements (on the order of 10 μm). These displacements are monitored using ultrasonic correlation-based and phase-shift methods (i.e., normalized cross-correlation, Kasai's 1-D, and Loupas' 2-D autocorrelators) [11], [13]–[15]. ARFI images are generated by laterally translating the ultrasonic beams and estimating the displacements through time at each location [11]. This scheme generally includes between 50 and 120 A-line acquisitions at 10 to 60 lateral locations, generating large amounts of data (1 to 50 MB). Data processing, including cubic spline interpolation, displacement estimation, and motion filtering, has traditionally been performed on a single CPU, taking from several seconds to more than a minute to process the data, depending on the size [14]. Real-time estimation and display of ARFI images would be advantageous in the development of new algorithms and sequences as well as for clinical feasibility studies. In this paper, GPU parallel processing is developed and implemented for ARFI imaging.

Manuscript received June 29, 2010; accepted October 25, 2010. This work was supported by NIH grants R01 EB001040, R01 EB002312, and CA142824, the Coulter Foundation, and the Duke University Department of Anesthesiology.

The authors are with the Department of Biomedical Engineering, Duke University, Durham, NC (e-mail: stephen.rosenzweig@duke.edu).

Digital Object Identifier 10.1109/TUFFC.2011.1817

II. Background

A. NVIDIA Quadro FX 3700M and CUDA

The NVIDIA (Santa Clara, CA) Quadro FX 3700M is a high-end laptop graphics card, utilized in this case as part of a mobile workstation. The card's architecture consists of 16 multi-processors (MPs), each containing 8 stream processors for a total of 128 stream processors operating at 1.40 GHz. The Quadro FX 3700M has a total of 1 GB memory divided into 16 kB shared memory (SM) per MP and 64 kB constant memory, with the rest being global memory; each MP also has access to 8192 registers.

One manner in which CUDA can be used is as an extension of the C and C++ languages, primarily allowing for memory copies between the CPU and GPU memory banks and the ability to launch programs, known as kernels, on MPs. Each kernel consists of a grid of blocks, where a block is a group of threads. Each block is launched on a single MP, such that all the threads in a block run in parallel. The block size is limited to 512 threads and the grid size is limited to 2^{32} blocks. When a block of threads is sent to a MP, a SIMT (single-instruction, multiple-thread) unit divides the block into groups of 32 threads called warps and

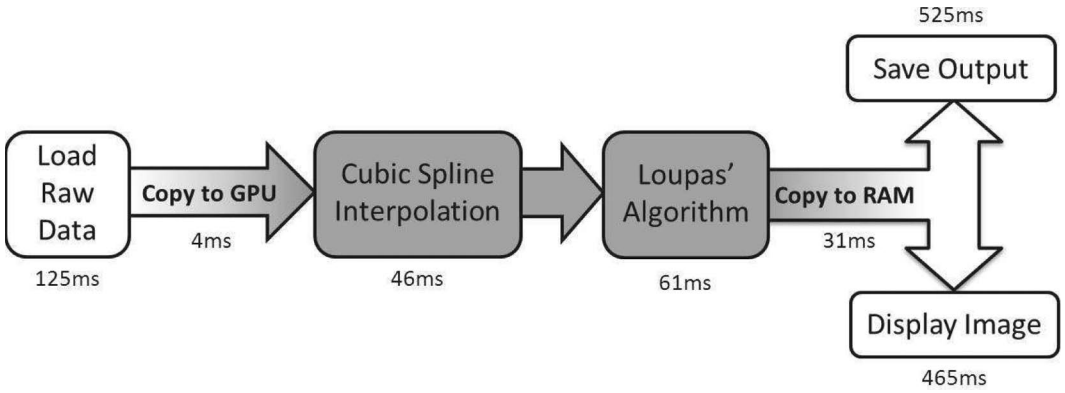

Fig. 1. A flow diagram of the overall algorithm is shown along with the computation time for each step when using the CUDA code. The CPU stages of the program are shown in white and the GPU processing steps are shaded. The time required to load the data, copy it to the graphics card, process it, and copy it back to RAM is less than that to either save the output or display an image of the data.

the compiler, SIMT unit, and internal thread scheduler control when each thread is sent to a stream processor.

B. Acoustic Radiation Force Impulse (ARFI) Imaging

Acoustic radiation force arises from a transfer of momentum from an ultrasonic wave to the medium through which it is traveling [16]. This momentum transfer is due to both absorption and scattering of the wave, and is directly related to the acoustic attenuation [16], [17]. ARFI imaging utilizes this acoustic radiation force by applying short duration (<1 ms) focused ultrasound pushing pulses that displace tissue [17]. Typical ARFI images are generated by acquiring at least one conventional reference A-line at the region of interest, then applying the pushing pulse, and finally acquiring additional tracking A-lines. The data can be acquired either in RF format or as quadrature demodulated (IQ) data. The response of the tissue is determined by estimating the displacement of the tissue between the pre-push reference and the post-push tracks [17]. The displacement estimation process for IQ data involves two standard ultrasonic image processing steps: cubic spline interpolation and Loupas' 2-D autocorrelator [13], [14]. In this paper, we present GPU-based algorithms that we have developed to perform the sequence of operations shown in Fig. 1.

III. Methods

A. Cubic Spline Interpolation

Cubic spline interpolation is used extensively in ultrasonic tracking methods to upsample the data to improve the precision of the velocity or displacement estimates [14]. Spline interpolation represents an extension of linear interpolation such that the first- and second-order derivatives are continuous. For a given function $y_i = y(x_i)$, $i = 1 \ldots N$, the interpolating spline for $x \in [x_i, x_{i+1}]$ is given by

$$S(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad (1)$$

where b_i, c_i, and d_i are coefficients that are computed by solving a tridiagonal matrix [18]. Once the coefficients are computed, the abscissae at which the data are to be interpolated are substituted into (1).

The traditional solution to a tridiagonal matrix, such as that used to solve for the spline coefficients, involves a decomposition loop and a backsubstitution loop, both of which iterate over every (x_i, y_i) [18]. This algorithm is not well suited to parallelization because the solution is dependent upon all of the data points. To solve for the coefficients using a parallel GPU architecture, a modified implementation was developed by dividing the long vector of data into small overlapping subsets.

The problem was divided into N/n spline interpolations, where n is the number of coefficients calculated in each interval and N is the total number of points to be interpolated. The spline coefficient kernel was then launched with each thread solving the tridiagonal matrix associated with $n + 2k + 1$ data points, where k is the number of overlapping points. The data were first copied into shared memory such that thread 0 copied $(y_0, \ldots, y_{n+2k})$, thread 1 copied $(y_n, \ldots, y_{2n+2k})$, and so forth. Each thread then computed the $n + 2k + 1$ coefficients associated with the data that it copied over using the traditional tridiagonal matrix solution. The coefficients were copied back into global memory so that the first k and last $k + 1$ coefficients were thrown out. Thus, thread 0 copied $(b_0, \ldots, b_{n+k-1})$, thread 1 copied $(b_{n+k}, \ldots, b_{2n+k-1})$, thread 2 copied $(b_{2n+k}, \ldots, b_{3n+k-1})$, and so forth, into global memory.

The splines were then evaluated by launching a new kernel and having each thread of each block copy a y_i, b_i, c_i, and d_i into shared memory to compute $S(x)$ for $x \in [x_i, x_{i+1}]$. The output was then directly written to global memory. The speed and accuracy of the GPU-based spline interpolation was compared with the traditional implementation while varying the number of points per thread, the number of overlapping points, and the number of points to be interpolated.

B. Loupas' Phase Shift Estimator

One algorithm that is often used in displacement estimation for ARFI images is presented by Loupas *et al.* and

$$\bar{u} = \frac{c}{4\pi f_c} \frac{\arctan\left(\frac{\sum_{m=0}^{M-1}\sum_{n=0}^{N-2}[Q(m,n)I(m,n+1)-I(m,n)Q(m,n+1)]}{\sum_{m=0}^{M-1}\sum_{n=0}^{N-2}[I(m,n)I(m,n+1)+Q(m,n)Q(m,n+1)]}\right)}{1+\arctan\left(\frac{\sum_{m=0}^{M-2}\sum_{n=0}^{N-1}[Q(m,n)I(m+1,n)-I(m,n)Q(m+1,n)]}{\sum_{m=0}^{M-2}\sum_{n=0}^{N-1}[I(m,n)I(m+1,n)+Q(m,n)Q(m+1,n)]}\right)/(2\pi f_{\mathrm{dem}})} \tag{2}$$

utilizes IQ data [13], [14]. The algorithm is based on (2), see above, where M is the axial averaging range, m is the axial sample being used, N is the ensemble length, n is the IQ line being used, c is the speed of sound, f_c is the center frequency, f_{dem} is the demodulation frequency, and $\bar{u}$ is the average displacement in that axial range. For ARFI imaging, a single pre-push reference is typically compared with each of the tracks to estimate the displacement through time and depth. Thus, the ensemble length is always 2, and for a given lateral location, the reference IQ line is constant.

To implement (2) using CUDA, each upsampled IQ line (2465 points) was divided into 512-point sections, corresponding to the number of threads in a block. Each section overlapped by M points, the axial averaging window size, with the previous section so that the displacement would be estimated for each point in the IQ line. The grid of blocks was then created with the number of blocks equaling the total number of sections including all lateral locations and tracks per location. The kernel begins by declaring six 512-point shared memory arrays (12 kB) such that 4 arrays correspond to the I and Q components of one section of a paired reference and track, with the other arrays being used to store the data from the summations in (2).

To parallelize the algorithm, each thread computes a single parameter inside the summations of (2). Once the parameters are saved in shared memory, each thread performs a summation so as to have the minimum number of divergent threads. The parameters are computed such that shared memory blocks are overwritten so that all of the parameters are computed within the 12 kB that was originally declared. After the four summation parameters are computed, the demodulation frequency vector is copied from GPU global memory and equation (2) is evaluated using the built-in *atan2f* function. The displacement value is written directly to GPU global memory.

C. Computational Speed Tests

All code was compiled on a Dell Precision M6400 laptop (Dell Inc., Round Rock, TX) with 4 GB RAM that was running Microsoft Windows XP (Microsoft Corp., Redmond, WA). The laptop had an Intel Core 2 Extreme Q9300 CPU (Intel Corp., Santa Clara, CA) operating at 2.53 GHz and an NVIDIA Quadro FX 3700M with 16 multi-processors and 1 GB total memory. The CUDA code (GPU/CPU) was compiled using the nvcc compiler for CUDA 2.3 and the traditional C++ code (CPU only) was compiled using the gcc compiler for speed comparisons.

The programs were tested on 10 independent data sets acquired from a modified Siemens Sonoline Antares scanner (Siemens Medical Solutions USA, Inc., Ultrasound Division, Mountain View, CA) using a VF7–3 transducer operating at 5.33 MHz. The data sets have 52 total push locations, 80 track pulses per push, and 493 I and Q samples per track. Each data set was processed 20 times and the run times were averaged. The error bars in all of the figures show the standard deviation of the average computation time between the 10 data sets. The computation times and speed increases stated for cubic spline interpolation include copying the raw 16-bit integers from RAM to the GPU global memory and for Loupas' algorithm include copying the displacement estimates from the GPU global memory back into RAM.

The multi-processor occupancy for algorithms that were executed on the GPU was also computed. The occupancy is a ratio of the number of active warps on a MP to the maximum possible number of active warps, which is 24 for the NVIDIA Quadro FX 3700M. The occupancy is a measure of the efficiency of the code, with the most efficient code having an occupancy of 1 [1].

IV. Results

Fig. 2 compares the computation time with the number of overlapping points (k) and points per thread ($n + 2k + 1$) for interpolating 2×10^6 points from a sampling rate of 8.9 to 44.4 MHz. The computation time is directly proportional to the number of overlapping points and has a more complicated dependence on the number of points per thread. The error associated with the parallelized approximation of the cubic splines as compared with the traditional algorithm was also computed. The maximum rms errors were 0.33%, 0.15%, 0.12%, and 0.12% for 1, 2, 3, and 4 points of overlap, respectively.

A further analysis of the speed of the CUDA code was performed using 15 points per thread and 2 points of overlap. The CUDA code was compared with a traditional cubic spline interpolation algorithm [18] as programmed in C++. The number of points that were interpolated was varied logarithmically between 1×10^4 and 2×10^6 points. The resulting time to perform interpolation is shown in Fig. 3(a) and the speed increase of the CUDA code over the C++ code is shown in Fig. 3(b).

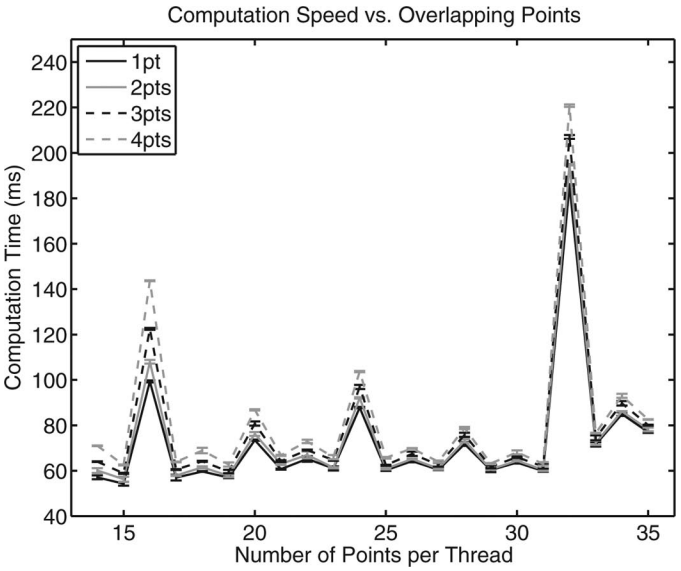

Fig. 2. The computation time of cubic spline interpolation using CUDA as a function of number of points per thread and number of overlapping points. The computation time is linearly related to the number of overlapping points and is a complicated function of the number of points per thread that should be optimized empirically.

Using these outlined algorithms, the computation time for the CUDA code was compared with previously optimized C++ code [14]. The speed increase for both the interpolation and Loupas' algorithm are shown in Figs. 4(a) and 4(b). In Fig. 4(a), the speed increase is shown as a function of the number of track pulses used, assuming 52 push locations, and Fig. 4(b) assumes 80 track pulses while varying the number of push locations. The MP occupancy was 0.083, 0.667, and 0.667 for the spline coefficient, the spline evaluation, and the Loupas' algorithm kernels, respectively.

The speed increase of the CUDA code is constant for greater than 40 tracks (assuming 52 push locations) or greater than 20 push locations (assuming 80 tracks). In these cases, the CUDA implementations are 41× and 27× faster for cubic spline interpolation and Loupas' algorithm, respectively. The interpolation time includes copying the raw data to the graphics card, and the computation time for Loupas' algorithm includes copying the displacement estimates back to RAM. Additionally, the displacements estimated with the CUDA code had a maximum rms error of 0.012 μm (1.1%) across the 10 data sets compared with the C++ code.

Fig. 5 shows the computation time associated with the components of processing an ARFI data set, including: acquiring the raw data, estimating the displacements, saving the displacement values to disk, and displaying an image. The data acquisition duration is based on transmitting 80 tracking pulses per location with a pulse repetition frequency of 7 kHz at 13 lateral locations using 4:1 parallel tracking for a total of 52 lateral locations. The data are saved using *fwrite* on single-precision floating point numbers, and the graphical display is accomplished with OpenGL.

V. Discussion

The goal of this work is to develop a GPU-based real-time ARFI data processing system for use in clinical feasibility studies of ARFI imaging. The two principle computational steps required to achieve this goal are data interpolation and displacement estimation. To develop efficient GPU code for these algorithms, the two primary considerations were parallelizing the algorithms so that each data point is independent and reducing the computational memory footprint of the data to at most 16 kB per 512 data points.

As shown in Fig. 2, the computation time for spline interpolation increases monotonically with the number of overlapping points, but has a complicated dependence on the number of points per thread. The monotonic behavior

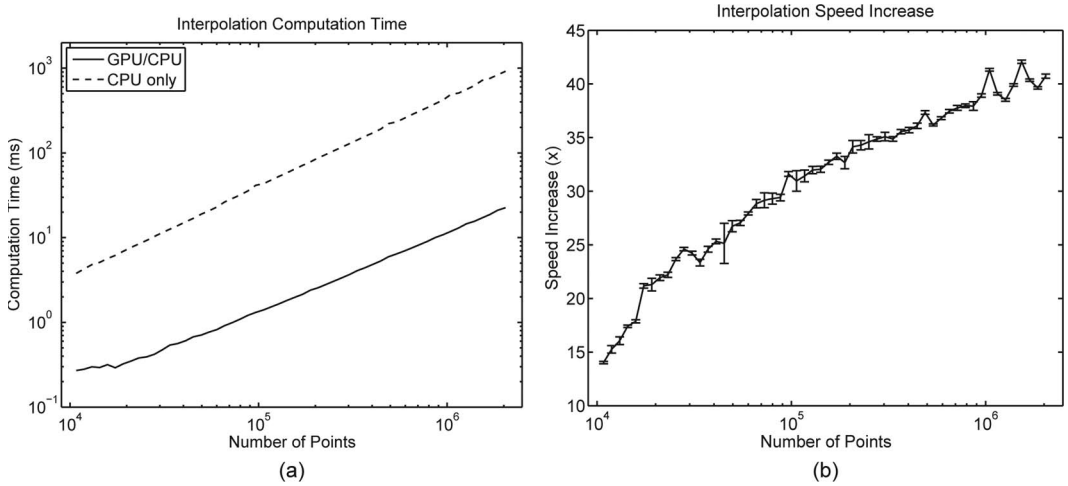

Fig. 3. The computation time (a) and speed increase (b) of cubic spline interpolation as a function of the number of points to be interpolated. The efficiency of the CUDA code increases as the number of points to be interpolated increases, eventually plateauing at approximately 41× faster than the C++ code.

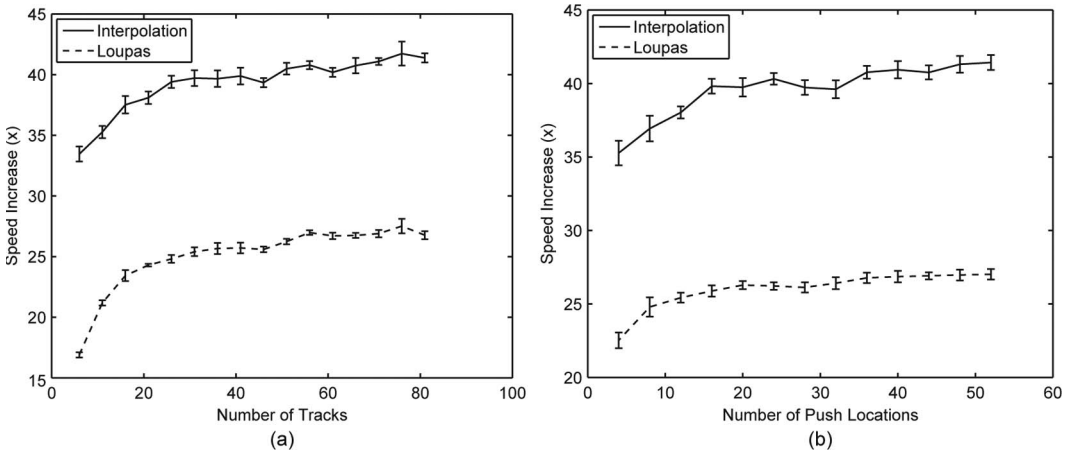

Fig. 4. Speed increase for cubic spline interpolation and Loupas' algorithm as a function of (a) the number of track pulses (assuming 52 push locations) and (b) the number of push locations (assuming 80 track locations). As in Fig. 3(b), the speed increase of interpolation plateaus at approximately 41× faster for the CUDA code as compared with the C++ code. Similarly, Loupas' algorithm plateaus at 27× faster for the CUDA code. Additionally, 37% of the computation time associated with Loupas' algorithm is devoted to copying the displacement estimates from GPU memory to CPU memory.

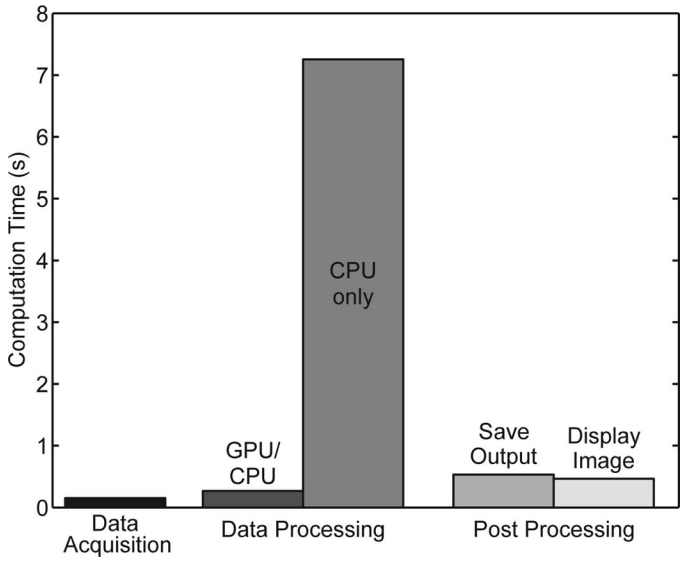

Fig. 5. Time required for acquiring the raw data, estimating the displacements, saving the estimates to disk, and displaying an image using OpenGL. The time required for data processing is significantly reduced by using CUDA (i.e., GPU/CPU versus CPU only), and therefore, it is no longer the rate limiting step in generating ARFI images.

is expected because of the increasing number of computations required; however, the other trend is much more difficult to analyze. There is a small increase in the computation time as a function of the number of points per thread, but the dominant behavior seen is the erratic behavior from point to point. It is hypothesized that these rapid variations are due to the number of bits being copied between global and shared memory and the way the individual threads are accessing the data in shared memory. It is not feasible to account for all of these factors when initially developing the code, and therefore, optimization must be performed empirically by testing different numbers of points per thread.

Figs. 3(a) and 3(b) show the increasing efficiency of the CUDA code as the number of interpolation points is increased. This increase in efficiency is due to two major effects; first, as the number of points increases, the CUDA code can occupy more of the MPs. The second effect is that as the number of points to be interpolated increases, a greater portion of the computation time is spent computing the required values compared with copying the data from RAM.

With respect to displacement estimation, both cubic spline interpolation and Loupas' algorithm reveal a distinct trend of increasing efficiency as the number of total computation points is increased, ultimately leveling off, as shown in Fig. 4(a) and 4(b). Although efficient, neither of these algorithms is perfectly optimized, as seen by the MP occupancy for each algorithm, indicating that the MP is not being used to its maximum efficiency. In all cases, the occupancy is not equal to 1, but as seen in Fig. 5, the data processing step is no longer the rate-limiting step, and thus additional optimization of the algorithms would not greatly improve the overall speed.

The speed increase of Loupas' algorithm is much smaller than that of interpolation primarily because of the memory copy operation back to RAM. This discrepancy results from the raw IQ data being 16-bit integers, which are then upsampled and stored as single-precision floating point numbers (floats). After the displacements are estimated, they are also stored as floats at the high sampling frequency. Thus, the number of bytes copied to the graphics card memory is only 20% of the number copied back to RAM. The second copy operation accounts for 37% of the computation time of Loupas' algorithm. If this copy time was not included, the speed increase for Loupas' algorithm would be almost identical to that of the interpolation.

The small difference in the value of displacement estimates between the C++ code and CUDA code has two sources: the approximations made in the cubic spline in-

terpolation and the use of single-precision floating point numbers. Overall, the rms error in the displacement estimates was very small (0.012 μm, 1.1%) compared with the ARFI displacements induced (1 to 5 μm). This error is also an order of magnitude less than the Cramer-Rao predicted lower bound for the accuracy of the measurements [19].

The total time to read in the raw data from disk, estimate the displacements using CUDA, and copy the data back to RAM is 267 ms compared with 7255 ms to perform the same operations using C++, an overall speed increase of 27×. The data processing using CUDA does take longer than the data acquisition time (152 ms), but the displacement estimation is no longer the rate limiting step when either saving the output or displaying an image using OpenGL, which requires over 450 ms to initialize the graphics and display an image. Although the time required to display an image is relatively large, this can likely be reduced by using CUDA 3.0, which was recently released [20]. This version of CUDA allows for the use of OpenGL while data are still on the graphics card rather than necessitating a copy to RAM and then back to the graphics card.

VI. Conclusions

We have shown that two algorithms widely used in ultrasonic data processing, and specifically in ARFI imaging, are suitable for parallel execution on a GPU by demonstrating that the data points can be processed independently and that there is a maximum computational memory footprint of 16 kB per block of threads. Analysis of the performance shows speed increases of more than 40× for the algorithms and more than 27× including memory copy operations. Additionally, the error associated with using single-precision as well as the cubic spline interpolation algorithm is insignificant compared with the magnitude of ARFI displacements (<2%). We conclude that this data processing approach holds great promise for real-time display of ARFI data as well as for many other ultrasonic research applications.

Acknowledgments

Special thanks are given to Siemens Medical Solutions USA, Inc., Ultrasound Division for their technical assistance and to Dr. S. Grant, T. Milledge, and Dr. J. Pormann for their valuable insights.

References

[1] *NVIDIA CUDA Programming Guide v2.3.* NVIDIA Co., Santa Clara, CA, 2009.

[2] F. Xu and K. Mueller, "Real-time 3D computed tomographic reconstruction using commodity graphics hardware," *Phys. Med. Biol.*, vol. 52, no. 12, pp. 3405–3419, 2007.

[3] G. C. Sharp, N. Kandasamy, H. Singh, and M. Folkert, "GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration," *Phys. Med. Biol.*, vol. 52, no. 19, p. 5771–5783, 2007.

[4] Y. Guorui, T. Jie, Z. Shouping, D. Yakang, and Q. Chenghu, "Fast cone-beam CT image reconstruction using GPU hardware," *J. XRay Sci. Technol.*, vol. 16, no. 4, pp. 225–234, 2008.

[5] B. Jang, D. Kaeli, S. Do, and H. Pien, "Multi GPU implementation of iterative tomographic reconstruction algorithms," in *Proc. IEEE Int. Conf. Symp. Biomedical Imaging*, Piscataway, NJ, 2009, pp. 185–188.

[6] T. Schiwietz, T. Chang, P. Speier, and R. Westermann, "MR image reconstruction using the GPU," *Proc. SPIE*, vol. 6142, no. 1, art. no. 61423T, 2006.

[7] H. Guo, J. Dai, and J. Shi, "Fast iterative reconstruction method for propeller MRI," *Proc. SPIE*, vol. 7497, no. 1, art. no. 74972O, 2009.

[8] Q. Zhang, X. Huang, R. Eagleson, G. Guiraudon, and T. M. Peters, "Real-time dynamic display of registered 4D cardiac MR and ultrasound images using a GPU," *Proc. SPIE*, vol. 6509, no. 1, art. no. 65092D, 2007.

[9] M. Zhao and S. Mo, "A GPU based high-definition ultrasound digital scan conversion algorithm," *Proc. SPIE*, vol. 6509, no. 1, art. no. 76252M, 2010.

[10] L. Chang, K. Hsu, and P. Li, "Graphics processing unit-based high-frame-rate color Doppler ultrasound processing," *IEEE Trans. Ultrason. Ferroelectr. Freq. Control*, vol. 56, no. 9, pp. 1856–1860, 2009.

[11] K. R. Nightingale, M. Soo, R. Nightingale, and G. E. Trahey, "Acoustic radiation force impulse imaging: *In vivo* demonstration of clinical feasibility," *Ultrasound Med. Biol.*, vol. 28, pp. 227–235, 2002.

[12] K. R. Nightingale, M. L. Palmeri, and G. E. Trahey, "Analysis of contrast in images generated with transient acoustic radiation force," *Ultrasound Med. Biol.*, vol. 32, pp. 61–72, 2006.

[13] T. Loupas, J. T. Powers, and R. W. Gill, "An axial velocity estimator for ultrasound blood flow imaging, based on a full evaluation of the Doppler equation by means of a two-dimensional autocorrelation approach," *IEEE Trans. Ultrason. Ferroelectr. Freq. Control*, vol. 42, no. 4, pp. 672–688, 1995.

[14] G. F. Pinton, J. J. Dahl, and G. E. Trahey, "Rapid tracking of small displacements with ultrasound," *IEEE Trans. Ultrason. Ferroelectr. Freq. Control*, vol. 53, pp. 1103–1117, 2006.

[15] C. Kasai, K. Namekawa, A. Koyano, and R. Omoto, "Real-time two-dimensional blood flow imaging using an autocorrelation technique," *IEEE Trans. Sonics Ultrason.*, vol. 32, no. 3, pp. 458–464, 1985.

[16] G. R. Torr, "The acoustic radiation force," *Am. J. Phys.*, vol. 52, pp. 402–408, 1984.

[17] K. R. Nightingale, M. Palmeri, R. Nightingale, and G. Trahey, "On the feasibility of remote palpation using acoustic radiation force," *J. Acoust. Soc. Am.*, vol. 110, pp. 625–634, 2001.

[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Cambridge, UK: Cambridge University Press: *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. 1992.

[19] W. F. Walker and G. E. Trahey, "A fundamental limit on delay estimation using partially correlated speckle signals," *IEEE Trans. Ultrason. Ferroelectr. Freq. Control*, vol. 42, no. 2, pp. 301–308, 1995.

[20] *NVIDIA CUDA Programming Guide v3.0.* NVIDIA Co., Santa Clara, CA, 2010.

Stephen Rosenzweig was born in New York, NY, in 1986. He received the B.S.E. degree in biomedical engineering from Duke University, Durham, NC, in 2008. He is currently a James B. Duke and NIH Training Grant graduate fellow in the Biomedical Engineering department at Duke University.

Mark L. Palmeri received his B.S. degree in biomedical and electrical engineering from Duke University, Durham, NC, in 2000. He was a James B. Duke graduate fellow and received his Ph.D. degree in biomedical engineering from Duke University in 2005 and his M.D. degree from the Duke University School of Medicine in 2007. He is currently an Assistant Research Professor in biomedical engineering and anesthesiology at Duke University. His research interests include ultrasonic imaging, characterizing the mechanical properties of soft tissues, and finite element analysis of soft tissue response to acoustic radiation force excitation.

Kathy Nightingale received her B.S. degree (electrical engineering) in 1989 from Duke University. She served in the United States Air Force as a program engineer from 1989 to 1992. She received her Ph.D. degree in biomedical engineering from Duke University in 1997. Her research interests include the investigation of radiation force based imaging methods, elastography, finite element modeling of soft tissues, ultrasonic flow detection, and therapeutic ultrasound.