REAL-TIME PROCESSING IN DYNAMIC ULTRASOUND ELASTOGRAPHY:A GPU-BASED IMPLEMENTATION USING CUDA

Emmanuel Montagnon[†], *Sami Hissoiny*[‡], *Philippe Després*[§], *Guy Cloutier*[†,¥,*]

† Laboratory of Biorheology and Medical Ultrasonics, Université de Montréal Hospital Research Center (CRCHUM), Montréal, Québec H2L 2W5, Canada; and Institute of Biomedical Engineering, Université de Montréal, Montréal, Québec H3C 3J7, Canada
‡ Department of Computer and Software Engineering, ÉcolePolytechnique de Montréal, Montréal, Québec H3T 1J4, Canada
§ Department of Radiation Oncology, Université de Montréal Hospital (CHUM), Montréal, Québec H2L 4M1, Canada; now with the Department of Radiation Oncology, Centre hospitalier universitaire de Québec (CHUQ), Québec, Québec G1R 2J6, Canada
¥ Department of Radiology, Radio-Oncology and Nuclear Medicine, Université de Montréal, Montréal, Québec H3T 1J4, Canada
* Corresponding author: Dr Guy Cloutier, Director, Laboratory of Biorheology and Medical Ultrasonics, Université de Montréal Hospital Research Center (CRCHUM), 2099 Alexandre de Sève, room Y-1619, Montréal, Québec H2L 2W5

ABSTRACT

This paper addresses the computational cost of the normalized cross-correlation (NCC) algorithm in ultrasound elastography. Parallel implementations of the NCC algorithm based on multicore architectures and a graphical processor unit (GPU) are formulated and applied to radio-frequency (RF) data from dynamic elastography experiments. Compared to single computer processor unit (CPU) performances, results show that parallel implementation of the NCC algorithm allows speedups of less than 5 for multi-threaded execution on CPU and up to 85 using a GPU. Processing frame rates from 80 to 173 sec^{-1} have been achieved for large fields of view with good spatial resolution. The trade-off between accuracy, spatial resolution and computational cost in displacement estimation using the NCC algorithm therefore appears obsolete. Open source codes for implementing the NCC algorithm on GPU are made available at www.lbum-crchum.com.

Index terms: Ultrasound, elastography, displacement estimation, parallel computing.

1. INTRODUCTION

Elastography concerns the assessment of viscoelastic properties of soft tissues/materials in response to a mechanical excitation [1]. Induced displacement estimation is a critical step for static, steady-state and dynamic elastography; indeed displacements and their spatial or time derivatives are used either to compute induced strains [2, 3], or celerity of shear waves [4]. Therefore, accuracy of most of the inversion methods (e.g., direct inversion) directly depends on the displacement estimator performance [5].

In order to ultrasonically assess a structure motion or a fluid velocity, estimators have been formulated in both the time and phase domains. Phase domain estimators, using auto-correlation methods, have been shown to be less robust to noise than time-domain ones [6]. Among time domain estimators, the normalized cross-correlation (NCC) is considered the most robust algorithm [7] and is widely used in elastography [3, 8-11]. However, the computational cost of the NCC is seen as a major drawback, and the sum-squared differences (SSD) estimator has been presented as a good compromise between performance and computational efficiency [5].

The NCC algorithm is based on the maximization of a cost function (the cross-correlation coefficient) in order to track motion of scatterers between consecutive image frames. A reference signal is divided into a set of windows and each one is compared to a window of a subsequent signal [12]. Computation and maximization of the cost function for a unique window can be seen as the elementary brick of the whole displacement estimation since the procedure is iterated for all reference windows of acquired signals. In dynamic elastography, displacement estimation can easily require more than one million iterations of the NCC algorithm, leading to long computational times.

Among parallel computing platforms, the OpenMP application program interface consists in a set of compiler directives and library routines, which extends C/C++ and Fortran language to allow the development of multithreaded applications [13]. Parallel programming on graphical processor units (GPUs) has been widely used in recent years in various scientific domains [14-16], and more specifically in ultrasonic topics such as beamforming[17] or simulations [18]. One of the most used platforms in GPU computing is the Compute

Unified Device Architecture (CUDA, version 3.1) developed by NVIDIA (Santa Clara, CA, USA), which is an extension of C programming.

In this paper, the NCC algorithm is implemented on a GPU using CUDA and on CPUs using OpenMP, which are both supported by shared memory architectures. Since displacement estimation relies on independent NCC algorithm iterations, the problem considered appears to be highly parallelizable. The goal is not to introduce a new formulation of the cross-correlation algorithm nor to evaluate its reliability, but to assess speedups resulting from various parallel implementations for a given NCC formulation. This paper is organized as follows: first, parallel implementation of the NCC algorithm using OpenMP and CUDA are presented. Resulting time speedups are then compared to single processor performances. Finally, results obtained from both platforms are discussed and perspectives arising from this work are presented.

2. METHODS

2.1 Normalized cross-correlation coefficient and parabolic interpolation:

The NCC algorithm is presented in Fig. 1. The NCC coefficients were computed from a reference window W_1 and various positions of the comparison window W_2. Defining s as the maximum shift between W_1 and W_2 according to the position of W_1, comparison windows are moved from $-s$ to $+s$ along the comparison signal. For each position of the reference window, the NCC coefficients given by (1) are computed leading to a $2s+1$points cross-correlation function,

$$R_{12}(j)=\frac{\sum\left[\left(W_1-\overline{W}_1\right)\left(W_2-\overline{W}_2\right)\right]}{\left[\sum\left(W_1-\overline{W}_1\right)^2\sum\left(W_2-\overline{W}_2\right)^2\right]^{\frac{1}{2}}}, \qquad (1)$$
$$j=\{-s,-s+1,\ \ldots\ ,s-1,s\}$$

where $\overline{W}_1$ and $\overline{W}_2$ are means of W_1 and W_2 respectively, and τ is the temporal shift between the two signals. Since the computed cross-correlation function is discrete, the maximum is estimated using a parabolic interpolation:

$$\tau_0=i+\frac{R_{i-1}-R_{i+1}}{2R_{i-1}-4R_i+2R_{i+1}} \qquad (2)$$

where i is the position of the maximum, R_i is the maximum of the cross-correlation function and R_{i-1}, R_{i+1} are points adjacent to R_i. Other interpolation methods have been proposed to avoid some false-peaks or aliasing issues. Those methods are not discussed here and the reader is referred to [19]. Physical displacements d are finally estimated using:

$$d=\frac{c\tau}{2} \qquad (3)$$

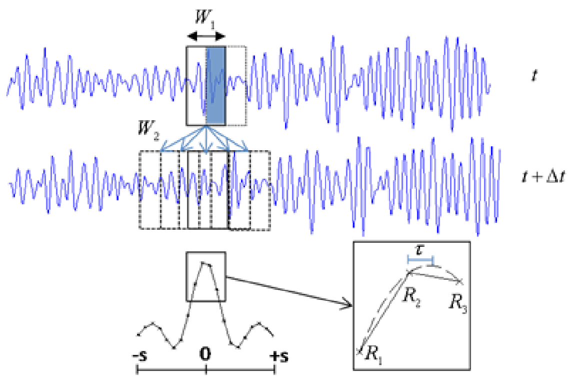

Figure 1. Normalized cross-correlation algorithm. From two consecutive signals acquired at two distinct times, the best match between ***W₁*** and multiple comparison windows ***W₂*** is found by maximizing the cross-correlation function. Parabolic interpolation is realized around the discretely computed cross-correlation function maximum to increase estimation accuracy.

where c = 1540 m/s is the celerity of sound in the case of soft tissues. In order to increase the spatial discretization, an overlap is applied to consecutive reference windows. For a given length of the reference signal, the number of NCC algorithm iterations depends on the window length and applied overlap.

2.2 Hardware architecture:

In this work, an Intel Core i7 CPU with a quad core supporting hyperthreading was used, running at 2.67 GHz, with Windows Vista 64-bits. The hyperthreading term means that each physical core is seen by the operating system as two logical processors, resulting in eight logical processors. GPU parallel computing was realized using a Nvidia GTX 480 graphic board and CUDA 3.1. Visual Studio Professional 2008 was used for all application developments, using a 64-bits compiler.

2.3 Serial implementation:

The single CPU implementation of the NCC algorithm was defined as the reference code for the rest of this paper (i.e., comparison basis for speedup performance assessments). Serial execution of the algorithm implies three nested for loops: along the time dimension (outermost loop: number of frames to process), along each reference signals (number of windows per radiofrequency signal, *i.e* RF-line) and finally along the $2s+1$ points to compute the cross-correlation function (see Fig. 2).

2.5 OpenMP implementation:

The parallel computing of the NCC algorithm using OpenMP was implemented with the omp pragma parallel for directive [20]. Block instructions contained between the pragma brackets were executed by multiple threads.

The number of threads was defined as the number of processors available on the The pragma definition requires specifying which variables are shared by all threads, or private to each thread. Reference and comparison RF-lines, window length, maximum overlap shift, and result pointer were defined as shared variables. The position of the reference window along the RF-line was defined as a private variable. Parallelization was realized at the second innermost for loop: the NCC algorithm was therefore iterated simultaneously by all threads until all the reference windows were processed. Since each thread writes results in distinct positions in the result array, no thread synchronization was necessary.

2.6 GPU implementation:

To allow computation of displacements on a GPU, RF-data must be copied onto the device's global memory. Parallel programming using CUDA introduces thread hierarchy. A grid is defined as a set of blocks, which are groups of threads. The kernel, which is a function executed on the grid, implemented the NCC algorithm. Therefore, one kernel processed one reference window and computed the displacement at one position and at one time. The block's dimension was set as the number of reference windows along a RF-line (see the pseudo codes in Fig. 2). The number of blocks within the grid was the number of RF-lines to be treated (1000 in this study). On the GTX480 graphic card, grid dimensions are limited to 1024 threads per block and to 65535 blocks per grid dimension. In the various configurations studied in this paper, the number of threads per block was always consistent with grid size limitations. Therefore, with the GPU implementation, the two outermost for loops were unrolled and replaced by only one computational grid, increasing the level of parallelization. The innermost for loop took place in the kernel itself. Once the GPU processing was completed, results were transferred from the GPU to the host (CPU).
Graphic cards are made of different kinds of memory: the global memory, which is large (1.5 GB on the GTX480 board used) but suffers from high latency, and smaller on-chip memory such as the shared memory (48 kB on the GTX480). The shared memory is distributed between all the threads of one block. However, the shared memory is limited in size and its use may require some additional programming work to formulate the problem accordingly. In order to take advantage of the on-chip memory, two consecutive RF-lines were transferred from the global memory to the shared memory. To reduce the transfer time, all threads of a block loaded a small part of the RF-lines on the shared memory, and all subsequent readings in a kernel accessed the shared memory. The parallel implementation on GPU using the global memory exclusively or the global plus shared memory are referred in the rest of this paper as CUDA I and CUDA II implementations, respectively.

```
for  (i=0 ,i<Nb_RF_Lines,i++)
  {
   for
(j=0,j<Nb_Windows,j++)
      {
       for (k=-s,k<(s+1),k++)
         { NCC coefficient
computation; }
       Cross-correlation
maximum search;
       Parabolic interpolation
       }
   }
```

(a)

```
for  (i=0,i<Nb_RF_Lines,i++)
  {
   #pragma omp parallel
     shared() private()
   {
     #pragma omp for
    for (j=0,j<Nb_Windows,j++)
     {
      for (k=-s,k<(s+1),k++)
        { NCC coefficient
computation; }
       Cross-correlation
maximum research;
       Parabolic interpolation
       }
    }
  }
```

(b)

```
//Sizing computational grid
dimGrid=Nb_RF_Lines;
dimBlock=Nb_Windows;
// grid execution
NCC_kernel<<<dimGrid, dimBlock>>>
(.....)

// kernel pseudo code
NCC_kernel(...)
{
 for (k=-s,k<(s+1),k++)
     { NCC coefficient computation; }
   Cross-correlation maximum research;
   Parabolic interpolation
     }
}
```

(c)

Figure2. Pseudo codes of the different implementations of the NCC algorithm.(a) Serial, (b) multi-threaded implementation using OpenMP, (c) parallel processing with CUDA. Directive characteristics to each platform are in bold. Nb_RF_lines and Nb_Windows designate the number of processed RF-lines and number of reference windows along RF-lines, respectively.

2.7 Parameters selected for CPU and GPU implementations:

Since there is generally a trade-off between accuracy, spatial resolution and computational cost for displacement assessments [12], four distinct configurations were used (Table 1) to assess the impact of the maximum comparison window shift s, and of the number of reference windows over a RF-line, on the performance of CPU (single and multi-core processors) and GPU (CUDA I and II) implementations. The overlap between two consecutive reference windows was fixed at 95% to achieve good spatial discretization. Means and standard deviations of computational time in seconds were determined for each configuration, from 51 executions of each code (neglecting the first cache execution). Speedups were computed from the ratio:

$$speedup = \frac{T_{Serial}}{T_{Tested}} \quad (4)$$

with$T_{Serial}$and T_{Tested}being the computational times of the CPU serial and tested implementations, respectively. The subscript Tested refers to either OpenMP, CUDA I or CUDA II implementation.

2.8 Experimental setup:

Ultrasound RF-data in dynamic elastography were acquired from in-vitro measurements in agar-gelatin phantoms. The experimental setup was identical to the one presented in [21]; RF-lines were acquired along an 8 cm depth at a 3850 Hz frame rate using ECG-gating with a SonixRPechograph (model Sonix RP, Ultrasonix Medical Corporation, Burnaby, BC, Canada). Acquired RF data were 16 bits depth and stored in raw files. Computations of the NCC algorithm were performed on 1000 successive RF-lines acquired by one element of the probe (to extrapolate to all 128 elements of the L14-5/38 linear array transducer, a simple upper bound approximation of the computational time consists in multiplying the execution times by 128, see the discussion).

Table 1. Window lengths in mm and in number of sample points (pts), and maximum comparison window shifts (s in number of points.

Configuration	Window Length (mm / pts)	*s* (pts)
#1	5.78 / 300	10
#2	5.78 / 300	4
#3	1.93 / 100	10
#4	1.93 / 100	4

3. RESULTS

Mean computational times, standard deviations and speedups obtained for each configuration ran 51 times are presented in Table 2. Reported time values include processing, and data transfer to and from the global memory for GPU implementations.
As seen in Table 2, for all implementations except CUDA I, the maximum window shift of 10 samples gave the longest computation time for a given window length; the window length had less impact on the processing time. Speedups obtained with the multi-threaded execution using OpenMP were quite consistent for all configurations, with values between 4.23 and 4.55. GPU processing using only the global memory (i.e., CUDA I) presented the wider range of speedups that were highly dependent on the window length (from 15.67 for a window of 5.78 mm to 89.47 for the window of 1.93 mm). Using the shared memory (CUDA II) allowed important speedups for all configurations (from 75.56 to 83.56). For the longest window length (configurations 1 and 2), CUDA II was 5 times faster than CUDA I. However, for configurations 3 and 4 (window length of 1.93 mm), CUDA I performed similarly or better than CUDA II. This point is discussed in the next section. Mean data transfer times between GPU and CPU were 0.4 ms and 0.6 ms for window shifts s of 4 and 10 samples, corresponding to 6% and 8.9 % of the execution time, respectively.

4. DISCUSSION

From Table 2, one can see that the parallel computing using GPUs clearly outperforms CPU implementations, even without shared memory management. The small dependence of the processing time with the window length for serial, multi-threaded and CUDA II implementations can be explained by the increased computational cost for large windows that is balanced by the reduced number of windows to process. For all implementations, the computational time linearly depended on the maximum shift to compute the cross-correlation function; therefore, any *a priori* information about displacement amplitude can be efficiently used to adjust the maximum shift and hence reduce computation times.

Table 2. Means and standard deviations of computation times in seconds obtained for various implementations of the NCC algorithm, neglecting transfer times from CPU to GPU. Speedups are in parenthesis. The serial implementation is used as the reference for all comparisons.

Configuration	#1	#2	#3	#4
Serial	7.903 ± 0.025 (1)	3.400 ± 0.005 (1)	8.857 ± 0.021 (1)	3.862 ± 0.008 (1)
OpenMP	1.826 ± 0.023 (4.33)	0.804 ± 0.008 (4.23)	1.946 ± 0.072 (4.55)	0.863 ± 0.0083 (4.47)
CUDA I	0.500 ± 0.000 (15.81)	0.217 ± 0.000 (15.67)	0.099 ± 0.000 (89.47)	0.047 ± 0.000 (82.17)
CUDA II	0.097 ± 0.001 (81.47)	0.045 ± 0.001 (75.56)	0.106 ± 0.000 (83.56)	0.049 ± 0.000 (78.81)

Multi-threaded execution using the eight logical cores available exhibited speedups of less than 5. A speedup equals to the number of processors available on the platform is a theoretical limit, especially with processors featuring hyperthreading. Obtained speedups can thus be explained since, as stated before, the eight processors are not physical processors but the result of hyperthreading four physical cores. Moreover, the memory bandwidth is shared between cores and therefore affects performance. Varying the number of threads between 2 and 16 revealed that setting the number of threads to the number of cores gave the best results (exploratory tests not shown). Moreover, nesting 2 *pragmas omp parallel for*for the two outermost *for* loops did not significantly reduce computation times. Performances of the presented OpenMP implementation are limited by hardware capabilities rather than by the formulated parallelization of the NCC algorithm.

The equivalent speedups obtained for the smallest window of 1.93 mm (configurations 3 and 4) with CUDA I and CUDA II highlight the cost of transfer from the global memory to the shared memory. Indeed, data transfer in the shared memory is beneficial only if many accesses are expected in the rest of the kernel execution. For large windows, the use of the shared memory is appropriate, and the data transfer cost is highly compensated by the numerous memory accesses, explaining speedup differences between CUDA I and CUDA II for the first two configurations (window length of 5.78 mm). CUDA offers asynchronous execution of kernels and copies between the host and the device, using CUDA streams. This did not yield increased speedups in our implementations, most likely due to the short transfer times compared to grid execution times.

Configurations tested in Table 1 were highly challenging in a computational sense since large windows, an important image depth and a significant number of processed RF-lines were considered. If one extrapolates results obtained for one element of the transducer (see Table 2, CUDA II, configurations 1 and 2) to the 128 elements of the linear array probe, processing frame rates from 80 to 173 images / s would be achievable. Note that this is a lower bound in term of frame rate since additional parallelization and speedups may be feasible (we simply multiplied the computational times by 128 for this approximation). Therefore, for future experiments in dynamic elastography, trade-offs between estimation accuracy, spatial resolution and computational cost appear no longer valid. A real time implementation of the NCC algorithm seems feasible in the context of dynamic elastography (i.e., images acquired at a frame rate in the kHz and display at a lower frame rate).

Speedups obtained using GPU-based parallel processing highly depend on the achievable parallelization level and on the device memory management, as seen in Table 2. For example, in the context of parallel image processing, speedups varied from 8 for discrete cosine transform (DCT) to more than 200 for edge detection [22, 23], highlighting the effects of data type or algorithm dependency on obtained speedups. Performances reported in this paper demonstrate the relevance of GPU parallel processing in displacement estimations for ultrasound elastography. The implementation adopted in this study would also be suitable for other displacement estimators such as the sum-squared difference (SSD) and the sum-absolute difference (SAD) methods, by modifying only the kernel function.

5. CONCLUSION

In this paper, performances of various implementations of displacement estimation in the context of dynamic ultrasound elastography using the NCC algorithm have been presented. OpenMP is an easy way to take advantage of multi-core architectures, but allows only moderate speedups. Parallel processing using GPUs yielded important speedups of up to 80 times for the considered problem therefore appearing as the most suitable implementation for real-time displacements processing in elastography. For the hardware described in this article (Sonix RP scanner, Nvidia GTX 480 graphic board, Intel Core i7 CPU with a quad core supporting hyperthreading), the codes of each implementation can be found on-line at www.lbum-crchum.com (see publication section).

REFERENCES

[1] K. J. Parker, *et al.*, "A unified view of imaging the elastic properties of tissue," *J Acoust.Soc.Am.*, vol. 117, pp. 2705-2712, 2005.

[2] M. O'Donnell, *et al.*, "Internal displacement and strain imaging using ultrasonic speckle tracking," *IEEE Trans. Ultrasonics, Ferroelectrics and Frequency Control,*, vol. 41, pp. 314-325, 1994.

[3] J. Ophir, I. Cespedes, and H. Ponnekanti, "Elastography: a method for imaging the elasticity of biological tissues," *Ultrasonic Imaging,* vol. 13, pp. 111-134, 1991.

[4] M. Tanter, *et al.*, "Quantitative assessment of breast lesion viscoelasticity: initial clinical results using supersonic shear imaging," *Ultrasound Med Biol.,* vol. 34, pp. 1373-1386, 2008.

[5] F. Viola and W. F. Walker, "A comparison of the performance of time-delay estimators in medical ultrasound," *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on,* vol. 50, pp. 392-401, 2003.

[6] I. A. Hein and W. D. O'Brien, Jr., "Current time-domain methods for assessing tissue motion by analysis from reflected ultrasound echoes-a review," *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on,* vol. 40, pp. 84-102, 1993.

[7] W. F. Walker and G. E. Trahey, "A fundamental limit on delay estimation using partially correlated speckle signals," *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on,* vol. 42, pp. 301-308, 1995.

[8] M. Elkateb Hachemi, S. Callé, and J. P. Remenieras, "Transient displacement induced in shear wave elastography: Comparison between analytical results and ultrasound measurements," *Ultrasonics,* vol. 44, pp. e221-e225, 2006.

[9] A. Hadj Henni, C. Schmitt, and G. Cloutier, "Three-dimensional transient and harmonic shear-wave scattering by a soft cylinder for dynamic vascular elastography," *The Journal of the Acoustical Society of America,* vol. 124, pp. 2394-2405, 2008.

[10] C. Schmitt, A. Hadj Henni, and G. Cloutier, "Characterization of blood clot viscoelasticity by dynamic ultrasound elastography and modeling of the rheological behavior. ," *Journal of biomechanics. In Press, Corrected Proof.,* 2011.

[11] Y. Zheng, *et al.*, "Measurement of the layered compressive properties of trypsintreated articular cartilage: An ultrasound investigation," *Medical and Biological Engineering and Computing,* vol. 39, pp. 534-541-541, 2001.

[12] L. Jianwen and E. E. Konofagou, "A fast normalized cross-correlation calculation method for motion estimation," *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on,* vol. 57, pp. 1347-1357, 2010.

[13] D. Mallón, *et al.*, "Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface.* vol. 5759, M. Ropo, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 2009, pp. 174-184.

[14] J. D. Owens, *et al.*, "GPU Computing," *Proceedings of the IEEE,* vol. 96, pp. 879-899, 2008.

[15] S. Hissoiny, B. Ozell, and P. Despres, "Fast convolution-superposition dose calculation on graphics hardware," *Medical Physics,* vol. 36, pp. 1998-2005, Jun 2009.

[16] S. Hissoiny, B. Ozell, and P. Despres, "A convolution-superposition dose calculation engine for GPUs

Fast convolution-superposition dose calculation on graphics hardware," *Medical Physics,* vol. 37, pp. 1029-1037, 2010.

[17] N. Carl-Inge Colombo, "Digital beamforming using a GPU," 2009, pp. 609-612.

[18] T. Reichl*, et al.*, "Ultrasound goes GPU: real-time simulation using CUDA," Lake Buena Vista, FL, USA, 2009, p. 726116.

[19] L. Xiaoming and H. Torp, "Interpolation methods for time-delay estimation using cross-correlation method for blood velocity measurement," *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on,* vol. 46, pp. 277-290, 1999.

[20] OpenMP, "OpenMP API, http://openmp.org," 2005.

[21] A. Hadj Henni, C. Schmitt, and G. Cloutier, "Shear wave induced resonance elastography of soft heterogeneous media," *Journal of biomechanics,* vol. 43, pp. 1488-1493, 2010.

[22] J. Kong*, et al.*, "Accelerating MATLAB Image Processing Toolbox Functions on GPUs " in *GPGPU'10*, Pittsburg, PA, USA, 2010.

[23] Y. Zhiyi, "Parallel Image Processing Based on CUDA," 2008, pp. 198-201.