GPU-Based Elasticity Imaging Algorithms

Nishikant Deshmukh, Hassan Rivaz, Emad Boctor

Johns Hopkins University, 3400 N. Charles Street, Computer Science Department, 224 NEB, Baltimore, Maryland 21218, nishikant@jhu.edu

Abstract. Quasi-static elastography involves estimating the displacement field of the tissue undergoing slow compression. Since elastography is computationally expensive, many compromises have been made to perform it in real time. Parallelized algorithms and implementations for freehand palpation elastography are proposed in this paper to speed the computation. We present parallel implementations for Normalized cross correlation elastography algorithm and for Dynamic programming elastography algorithm. Both methods are implemented in CUDA® using Graphics Processing Unit (GPU). We are able to achieve above 300 images per second for NCC and around 30 images per second for DP elastography algorithm.

Keywords: Dynamic Programming, Normalized Cross Correlation, GPU, Parallelization, Ultrasound Elastography, Real time strain imaging.

1 Introduction

Elastography, the computation of the spatial variation of the elastic modulus of tissue, is an emerging medical imaging method with medical applications such as tumor detection [1].This paper focuses on static elastography, a well known technique that applies quasi-static compression of tissue and simultaneously images it with ultrasound. Through analysis of the ultrasound images, a tissue displacement map can be obtained [2, 3]. A least squares technique is then typically used to generate a low noise strain estimate from the displacement map [3].

In Ultrasound Elastography we initially take RF data sample with Ultrasound probe just above the surface of the tissue, we call this image as uncompressed image and the next sample we take by compressing the tissue and we call this image as compressed image. The resulting images are 2D matrices with columns representing the axial direction along the path of RF waves emitted out of the probe; we refer to it as RF lines.

In Normalized cross correlation (NCC) method we compare RF lines from compressed and uncompressed images for displacement in axial direction, since the pressure is applied in axial direction and also since resolution is superior in this direction. We select a vectors of size $1\times r$ from RF line in uncompressed image and vectors of size $1\times s$ from RF line in compressed image such that $s > r$. Each window has 85% overlapping over each other along the RF line. These computations for small windows are independent of each other and can be computed in parallel. After calculating NCC we apply cosine fit interpolation, to get more approximate values, in the same thread in which NCC was calculated reducing the need to spawn a separate thread. The parallel algorithm will be discussed shortly.

Dynamic Programming (DP) is a numerical method for fast numerical optimization of the algorithms which are computationally intensive and complex in implementation. The most popular technique in DP is the top down and bottom up approach in which, in terms of Finite Automata, the present state depends on the previous state. This approach is particularly good for algorithms involving energy or cost functions which depend on the result from the previous stage. DP is hard to parallelize because of its dependence on the data from previous steps. With the introduction of many cores GPU's by NVidia and flexibility of NVidia CUDA programming environment it has become increasingly easy to parallelize existing implementations by using these environment. We thought of parallelization of NCC and DP Elastography on this hardware and utilize the computation power of these fast emerging compute capabilities and help the cancer research efforts in detecting and monitoring of tumors in real time.

2 The Parallel Algorithms

2.1 Parallel Algorithm for Normalized Cross-correlation

In NCC, every search window for comparison of the *n* RF lines in uncompressed data set (window size *r*) to the *n* RF lines in compressed data set (window size *s*) is computed in separate threads on the GPU [5]. The number of sample points per RF line is *m*. These threads since doing the same work are run in parallel. After applying cosine fit method to get sub sample interpolation, the result is a single pixel which is independent of other interpolation output pixel from other threads [5]. This output data independence eliminates the need to perform thread synchronization which makes the task highly inefficient [5]. The calculation of echo-strain, by applying median filter and low pass filter, and strain estimation, using least square methods, is calculated per pixel with every GPU thread assigned to every pixel and this method again gives data independence in output data pixels which ensures that the write operation can be done by each thread without interference [5]. To increase the throughput we performed volume rendering by processing multiple images at the same time by combining them horizontally along the axial direction [5].

Depending on size of window and percentage of overlapping we get number of samples k in output image. So we spawn $k \times n$ number of threads to calculate strain which is also dimension of the output image. For volume rendering with p number of images the number of thread becomes $k \times n \times p$. We used fast Normalized cross correlation [10] given by

$$\gamma(u,v) = \sum_{x,y} \frac{\left[f(x,y) - f'_{u,v}\right]\left[t(x-u,y-v) - t'\right]}{\left\{\sum_{x,y}\left[f(x,y) - f'_{u,v}\right]^2 \sum_{x,y}\left[t(x-u,y-v) - t'\right]^2\right\}^{0.5}} \quad (1)$$

where f is the image and sums over (x, y) under t which is the matching template and t' is the mean of template and $f'_{u,v}$ is the mean of $f(x, y)$ in the region under the feature[10].

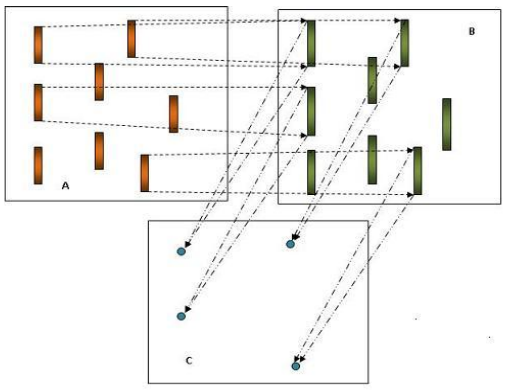

Fig. 1. A is uncompressed image, B is compressed image and C is output image. The windows in A and B are compared with each other giving out the pixel in C. All this comparisons are carried out in parallel on separate threads of GPU.

2.2 Parallel Algorithm for Dynamic Programming (DP) Elastography

DP Elastography is a known technique for calculating 3D echo strain image in medical imaging [4]. We focus on static elastography which applies quasi static compression of tissue [4]. DP Elastography is important since it has several advantages over the previous known elastography techniques. It reduces noise effect due to signal decorrelation between precompression and post-compression images and also reduces need for using large windows to reduce variance [4].

In 2D DP elastography [4] firstly we calculate the difference between two image signals g (i) and g' (i) along RF line and lateral direction.

$$\Delta(i, j, d_a, d_l) = \left| g_j(i) - g'_{j+d_l}(i + d_a) \right| \qquad (2)$$

Where $d_{a,\min} \le d_l \le d_{a,\max}$ is the displacement at sample *i* in axial direction (along the RF line) and $d_{l,\min} \le d_l \le d_{l,\max}$ is the displacement in lateral direction, and j = 1 to n and i = 1 to m, where m is the number of samples per RF-line and n is the number of RF lines [4].

A cost function is defined as

$$C_j(d_a, d_l, i) = \min_{\delta_a, \delta_l}\{\frac{C_j(\delta_a, \delta_l, i-1) + C_{j-1}(\delta_a, \delta_l, i)}{2} + \alpha R(d_a, d_l, \delta_a, \delta_l)\} + \Delta(d_a, d_l, i) \qquad (3)$$

Where $R(d_{a_i}, d_{l_i}, d_{a_{i-1}}, d_{l_{i-1}}) = (d_{a_i} - d_{a_{i-1}})^2 + (d_{l_i} - d_{l_{i-1}})^2$ is an axial and lateral direction smoothness regularization term, α is a weight for the regularization, j is the sample number for ith RF line, δ_a and δ_l are values that minimizes the cost function are stored for d_a, d_l and i. The cost function is minimized at i = m and the d_i values that have minimized the cost function are traced back to i = 1, giving the d_i for all samples. More information is in [4].

Our approach involves vertical partitioning of the input image of size $m \times n$ with a specific width w. The width can be set arbitrarily depending on the degree of parallelization needed. We prefer vertical partitioning since comparison is done along RF lines between two images and compression is applied along the RF lines. After dividing the image we get a new set of images which are (1/w) th size of the original image or in other words we get $\lceil n/w \rceil$number of subsets. The starting RF line of these subsets now acts as the pivot by applying independent cost function on them. On these subsets we apply the DP Elastography. But applying this approach will result in formation of straps like output since at the start of each subset; the application of cost function which does not depend on previous cost function will make an independent disparity from previous RF lines and these two disparities will vary greatly. To reduce this effect we propose the following approach.

Stage I: We calculate the disparity until a fixed depth d for every subset and preserve the cost function for this RF line.

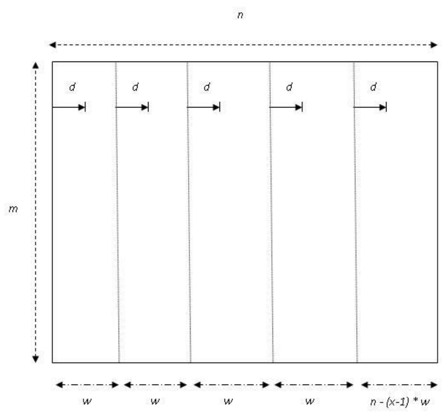

Fig. 2. Shows the Stage I, wherein we do DP steps until depth d at every interval of width w.

Stage II: Now considering the RF line in Step I at depth d for every subset as a new starting point and using its cost value as a pivot value we calculate disparity in both forward and backward directions (separate steps). We can say our original subset in Stage I got shifted right by distance d RF lines.

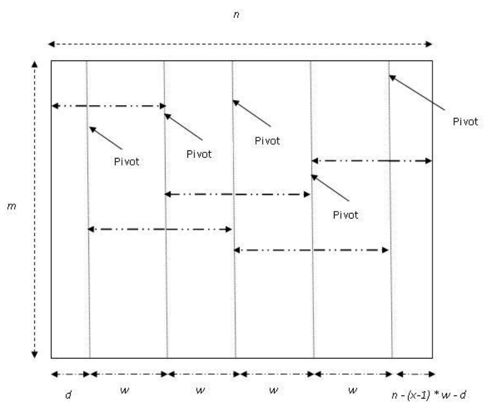

Fig. 3. Shows Stage II of DP Programming, after we have determined the new pivot values from Step I we apply DP algorithm forward and backward from these new points.

Stage III: Excluding the first subset and last subset, the intermediate subsets have two disparity values. We now calculate mean and weighted average of these disparities to get a fair disparity.

By using this approach we have successfully divided the data into $\lceil n / w \rceil$number of subsets giving us data independence. The forward and backward movement as in Stage II can be similarly run in parallel. The weighted average can be calculated independently for each pixel for output disparity. Coming back to the DP algorithm, the value of $\Delta(i, j, d_a, d_l)$ is independent for every value of i and j, and hence can be calculated in parallel with $m \times n \times d_a \times d_l$ threads. On GPU random memory access is very expensive incurring 400+ clock cycles [11], this latency can be effectively hided by spawning large amounts of thread. $C_j(d_a, d_l, i)$ can be calculated for every j value across all RF lines for i = 1 to m, we have to repetitively call the kernel in this loop since cost function depends on cost function of i -1. Hence the number of threads called by each kernel is $n \times \lceil n/w \rceil \times d_a \times d_l$. Finally the cost function minimization can be done for each individual subset in parallel in $\lceil n/w \rceil$ number of threads. To improve the performance we have used software managed caching feature of GPU.

To get maximum throughput we do multivolume processing of data by feeding multiple images for processing. So if numbers of streams are *p* then the overall threads will increase by multiplying *p* with number of threads in every stage.

3 Results and Discussion:

We are using NVidia Tesla C1060 GPU card for our experiments which has 240 Streaming Processor Cores and 4GB of DDR3 RAM.

3.1 Normalized Cross Correlation

In GPU the memory access is fastest in case of cached memory followed by sequential access and then the random memory access [9]. NCC involves accessing a window of size *r* and *s* in uncompressed and compressed image respectively along the RF lines. Because of the availability of small cache inside GPU we decided to approach this problem by performing serial access over the image. To do so we read in the input image in transpose forms making the RF line a row instead of column. In this way we perform serial access on the RF lines in each thread.

Table 1. Figure shows comparison timings of Normalized cross correlation method using CUDA with corresponding C implementation for $k = 139$ samples, $m=641$ and $n=73$.

	CUDA		**C**	
No. of images (*p*)	**time***	**time/image***	**time***	**time/image***
1	0.0035	0.0035	0.0324	0.0324
125	0.3776	0.0030	3.8171	0.0305
250	0.7696	0.0031	7.6362	0.0305
375	1.1267	0.0030	11.4495	0.0305
500	1.8251	0.0037	15.2657	0.0305
625	1.8723	0.0030	19.0773	0.0305
750	2.3079	0.0031	22.9005	0.0305

*All timings in seconds

Table 1 shows there is approximately 8X to 10X performance improvements in CUDA implementation compared to standard C implementation.

Table 2. Figure shows timing comparison for samples for corresponding number of frames.

	1	125	250	375	500	625	750
32	0.0044	0.4977	0.9987	1.4934	2.0052	2.4910	2.9969
61	0.0047	0.5607	1.1313	1.6846	2.2937	2.8096	3.3883
69	0.0053	0.6353	1.2808	1.9121	2.6222	3.1576	3.8247
139	0.0035	0.3776	0.7696	1.1267	1.8251	1.8723	2.3079
147	0.0053	0.5696	1.1512	1.7001	2.5980	2.8279	3.4453
192	0.0077	0.8683	1.7581	2.5812	3.9171	4.3041	5.2716

*All timings in seconds, horizontal heading shows number of images *p* and vertical shows number of samples(*k*).

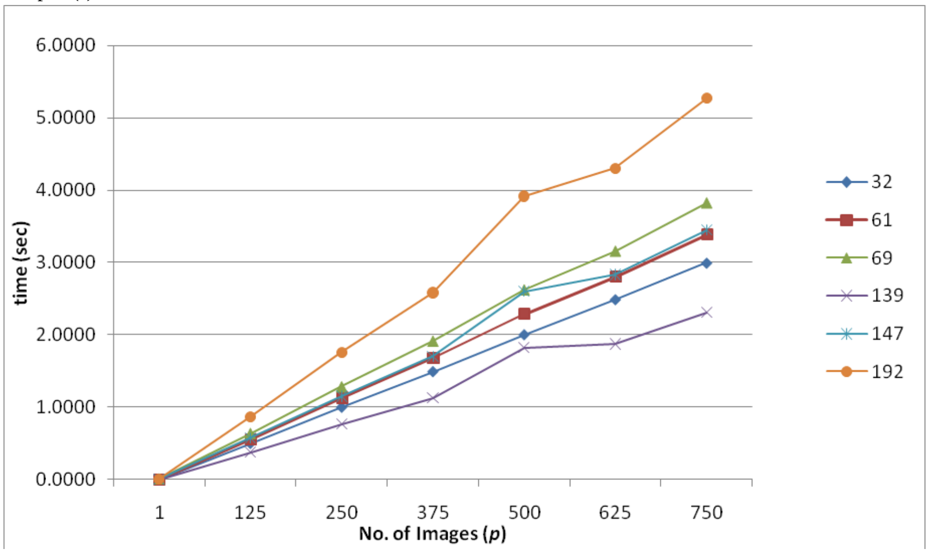

Fig. 4. Diagram shows plot for Table 2. We can see that we get optimal performance for k = 139 samples.

3.2 Dynamic Programming

GPU's global memory access is not through cache. So it incurs a memory access penalty of 400+ clock cycles for every Random access to the global memory [11]. It has a memory with small cache known as texture memory and additional constant memory with cache. Access to this constant memory cache is very fast [11] compared to the Global memory. Component $R(d_{a_i}, d_{l_i}, d_{a_{i-1}}, d_{l_{i-1}})$ to calculate C_j is calculated only once and is same for the whole image, according to (3) this component is used extensively. Hence we copied this array into constant memory and achieved 1.5X performance. We did not needed to read the input image in transpose form because from (2) calculation of $\Delta(i, j, d_a, d_l)$ is performed in separate threads which masks the memory access latency. We tested our program with CUDA profiler and found that all of our kernels are performing 0 non-coherent reads which means all our multiprocessors are accessing memory in coalesced form and all threads are accessing consecutive memory locations [11] which means we have a very efficient kernel design and configuration. More information on NVidia Cuda is in [8].

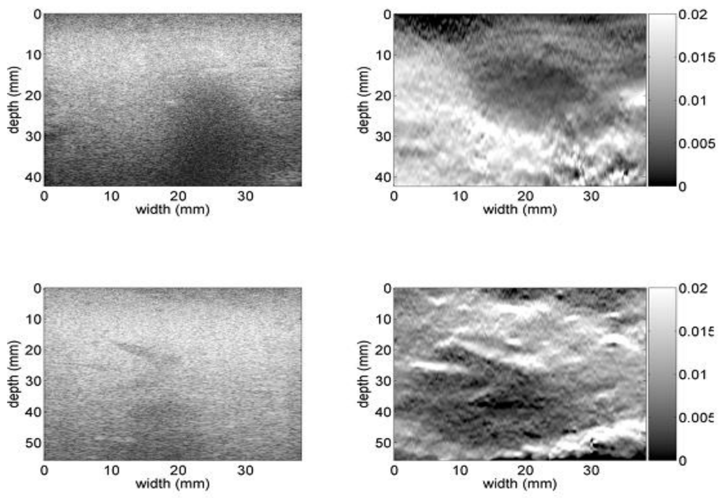

Fig. 5. On the right column, B-mode images of liver for two patients, and on the right column, the corresponding elasticity images are shown (from [6]).

Table 3. Table shows running DP code for different number of images in parallel and we can see that 200 images at an instance will give best performance.

	Timings in sec					
No. of Images	Stage I	Stage II forward	Stage II backward	Stage III	Total time	time/image
1	0.2860	0.4006	0.4294	0.0017	1.1177	1.1177
50	0.5866	0.8003	0.8743	0.0707	2.3319	0.0466
100	0.9427	1.2511	1.3934	0.1242	3.7114	0.0371
200	1.7547	2.1969	2.5555	0.2565	6.7636	0.0338
300	2.9633	3.5530	4.2858	0.3830	11.1852	0.0373
400	4.0215	5.0171	5.9079	0.5035	15.4500	0.0386
500	5.0205	6.3884	7.3990	0.6519	19.4599	0.0389

We ran our code for image of size m=1000 and n=100 with $d_a = 20$, $d_l = 2$, width w = 15 and depth d =10 and the result is in Table 3. We ran the code for C implementation for image of size m = 1000 and n=100 and the running is 0.2648 sec/image. From

Table 3 we can see that CUDA implementation gives improvement of approximately 6X – 8X compared to the corresponding C implementation.

4 Acknowledgements

The author would like to thank Mr. Matthew Bolitho, Dr. Philipp Stolka and Dr. Randal Burns for useful discussions.

5 Conclusion

GPU based parallelization provides opportunity for real time imaging and we have shown that we can successfully parallelize the present known techniques of Elastography for medical imaging. Sometimes it is important to expand the solution to get more opportunity for parallelization as seen in the case of Dynamic Programming Elastography. The results from Normalized Cross-correlation are encouraging with 300 images taking less than a second which is approximately 8 to 10 times the corresponding C implementation and Dynamic Programming based Elastography is having performance of 30 images per second which is 6 to 8 times the corresponding C implementation. GPU provides promising results but we have take in mind that memory access based computation can be performance degrading and special care needs to be taken to improve the performance of GPU based implementations using software managed caching and proper kernel configuration to get coalesced read.

6 References

[1] B. Garra, E. Céspedes, J. Ophir, S. Spratt, R. Zuurbier, C.Magnant, and M. Pennanen, “Elastography of breast lesions: Initial clinical results,” Radiology, vol. 202, pp. 79–86, 1997.

[2] J. Ophir, S. Alam, B. Garra, F. Kallel, E. Konofagou, T. Krouskop, and T. Varghese, “Elastography: Ultrasonic estimation and imaging of the elastic properties of tissues,” Annu. Rev. Biomed. Eng., vol. 213, pp. 203–233, Nov. 1999.

[3] J. Greenleaf, M. Fatemi, and M. Insana, “Selected methods for imaging elastic properties of biological tissues,” Annu. Rev. Biomed. Eng., vol. 5, pp. 57–78, Apr. 2003.

[4] Rivaz, H., Boctor, E., Foroughi, P., Zellars, R., Fichtinger, G., Hager, G., “Ultrasound Elastography: a Dynamic Programming Approach”, IEEE Trans. Medical Imaging Oct. 2008, vol. 27 pp 1373-1377

[5] E. M. Boctor, N. Deshmukh, M. S. Ayad, C. Clarke, K. Dickie, M. A. Choti, E. C. Burdette, "Three-dimensional heat-induced echo-strain imaging for monitoring high-intensity acoustic ablation", Vol. 7265, Medical Imaging 2009: Ultrasonic Imaging and Signal Processing.

[6] Rivaz, H., Fleming, I., Assumpcao, L., Fichtinger, G., Hamper, U., Choti, M., Hager, G., Boctor, E., "Ablation Monitoring with Elastography: 2D In-vivo and 3D Ex-vivo Studies", Medical Image Computing and Computer Assisted Intervention, MICCAI, New York, NY, Sept. 2008, pp 458-466

[7] J. Ophir, I. Cespedes, et.al. Elastography: A Quantitative Method for Imaging the elasticity of Biological Tissues. Ultrasonic imaging 1991; 13(2):111-34

[8] Cuda 2.2 Programming Guide and Reference Manual.

[9] GPU Gems 2, Chapter 32: Taking the Plunge into GPU Computing.

[10] J. P. Lewis, Fast Normalized Cross-Correlation.

[11] Victor Adrian Prisacariu, Ian Reid, fastHOG - a real-time GPU implementation of HOG, Technical Report No. 2310/09