A CPU-GPU HYBRID APPROACH FOR ACCELERATING CROSS-CORRELATION BASED STRAIN ELASTOGRAPHY

A Thesis

by

STHITI DEKA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2010

Major Subject: Electrical Engineering

A CPU-GPU HYBRID APPROACH FOR ACCELERATING CROSS-CORRELATION BASED STRAIN ELASTOGRAPHY

A Thesis

by

STHITI DEKA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Raffaella Righetti
Committee Members,	Deepa Kundur
	Jim Ji
	Riccardo Bettati
Head of Department,	Costas Georghiades

May 2010

Major Subject: Electrical Engineering

ABSTRACT

A CPU-GPU Hybrid Approach for Accelerating Cross-correlation Based Strain Elastography. (May 2010)

Sthiti Deka, B. Tech, National Institute of Technology, Tiruchirappalli

Chair of Advisory Committee: Dr. Raffaella Righetti

Elastography is a non-invasive imaging modality that uses ultrasound to estimate the elasticity of soft tissues. The resulting images are called 'elastograms'. Elastography techniques are promising as cost-effective tools in the early detection of pathological changes in soft tissues. The quality of elastographic images depends on the accuracy of the local displacement estimates. Cross-correlation based displacement estimators are precise and sensitive. However cross-correlation based techniques are computationally intense and may limit the use of elastography as a real-time diagnostic tool.

This study investigates the use of parallel general purpose graphics processing unit (GPGPU) engines for speeding up generation of elastograms at real-time frame rates while preserving elastographic image quality. To achieve this goal, a cross-correlation based time-delay estimation algorithm was developed in C programming language and was profiled to locate performance blocks. The hotspots were addressed by employing software pipelining, read-ahead and eliminating redundant computations. The algorithm was then analyzed for parallelization on GPGPU and the stages that would map well to the GPGPU hardware were identified. By employing optimization principles for efficient memory access and efficient execution, a net improvement of 67x with respect to the original optimized C version of the estimator was achieved. For typical diagnostic depths of 3-4cm and elastographic processing param-

eters, this implementation can yield elastographic frame rates in the order of 50fps. It was also observed that all of the stages in elastography cannot be offloaded to the GPGPU for computation because some stages have sub-optimal memory access patterns. Additionally, data transfer from graphics card memory to system memory can be efficiently overlapped with concurrent CPU execution. Therefore a hybrid model of computation where computational load is optimally distributed between CPU and GPGPU was identified as an optimal approach to adequately tackle the speed-quality problem in real-time imaging.

The results of this research suggest that use of GPGPU as a co-processor to CPU may allow generation of elastograms at real time frame rates without significant compromise in image quality, a scenario that could be very favorable in real-time clinical elastography.

To Ma, Papu and Shravan

ACKNOWLEDGMENTS

I am thankful to my graduate research advisor, Dr. Raffaella Righetti, for being supportive of my study, for giving me enough time to gather conclusions from various experiments, and most importantly, for giving me the liberty to experiment. Dr. Righetti's enthusiasm for research and recognition of any form of genuine effort from her students is endearing.

My thanks to Xu Yang for his invaluable contribution in speeding up the critical stages of the algorithm through implementation and analysis, to Raghavendra Desai whose initial efforts laid the foundation of the elastography software in C, to Anuj Chaudhry, Sanjay Nair, Srinath S. and Biren Parmar for their valuable inputs in delivering a stable software product.

I am grateful to Dr. Riccardo Bettati for his initial suggestions on considering GPGPU for acceleration at a time when we were looking at multi-threading for speed up. A major part of this work is realisation of his suggestion.

Thank you, Dr. Deepa Kundur, for your reassuring words everytime we met. Thank you, Dr. Jim Ji, for placing your trust in this work and for approving of any change in plan due to reassessment.

My thanks to Karen Dintino, Manager for Media Engineering group, AMD, for giving me the opportunity to study the CPU-GPU Fusion platform. I am thankful to my technical mentors, Hans Graves and Kishore Karighattam, for guiding me during the term. The co-op tour was an invaluable industry experience.

This work would not have been possible without the love and support of my friends and family. To them, I owe an immeasurable lot.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION 1

A. Motivation 1
B. Research Plan 3
1. Premise of This Thesis 3
2. Goals of This Research 4
3. Method and Preliminary Results 4
4. Impact and Significance of This Work 5
5. Structure of The Thesis 6

II ELASTOGRAPHY ON CPU 7

A. Principle of Elastography 7
B. Methodology 10
1. Acquisition of Tissue Motion Data 11
2. Time Delay Estimation for Axial Displacement 11
a. Computational Cost of Time Delay Estimation . . 14
3. Signal Conditioning 18
a. Computational Cost of Signal Conditioning 19
4. Interpolation 20
a. Parabolic Interpolation 20
b. Cosine Interpolation 21
c. Computational Cost of Interpolation 22
5. Median Filtering 22
a. Computational Cost of Median Filtering 23
6. Strain Estimation 23
a. Computational Cost of Strain Estimation 24
C. Image Quality Analysis 24
D. Performance Analysis 26

III PARALLEL ELASTOGRAPHY ON CPU-GPGPU 33

A. Parallel Computing on GPGPU 33
1. Elastography on GPU 36
B. Scientific Computing with GPGPU 37
1. CPU-GPU Communication and Control 39

CHAPTER Page

2. Hardware Architecture 40
a. Peak Theoretical GFLOPS and Bandwidth 43
3. Programming Model 45
4. Memory Model 46
5. Execution Model 48
C. Methodology 48
D. Results 54
1. Speed Up 54
a. GPU.Base 55
b. GPU.Occupancy 55
c. GPU.PinnedMemory 56
d. GPU.Coalesced 58
e. GPU.SharedMemory 58
f. GPU.ReducedDivergence 59
g. GPU.ReducedDataTrf 59
h. GPU.ParallelReduction 59
i. GPU.LoopUnroll 60
j. Hybrid Computation 60
2. Image Quality 63
a. Experimental Validation 69
E. Performance Analysis 70
F. Discussion 70

IV SUMMARY AND FUTURE WORK 80

A. Summary of Research 80
B. Avenues for Future Work 80
1. Improving CPU and GPU Utilization 81
2. Using Faster FFTW Libraries 81
3. Making the Algorithm Compute Bound 81
4. CPU-GPU Load Balancing 82
5. Integrated CPU-GPU 83
6. Speedup Using Multiple GPUs 84
C. Conclusion 84

REFERENCES 86

APPENDIX A 91

VITA 97

LIST OF TABLES

TABLE		Page
I	Hardware configuration of E4500 Intel Core2 Duo	27
II	Device attributes of 8800 GT	41
III	Device attributes of GeForce 8800 GT card	41
IV	Table of various memories on 8800 GT and their properties	49
V	Paged memory bandwidth on GeForce 8800GT	57
VI	Pinned memory bandwidth on GeForce 8800GT	57
VII	Summary of speed ups in axial strain elastography using different optimizations	62
VIII	Table of speed ups in GFLOPS different stages of elastography	65
IX	Table of CNRe of CPU and GPU computed elastograms when different background noise is applied	66
X	Table of SNRe of CPU and GPU computed elastograms when different background noise is applied	67
XI	Table of CNRe of CPU and GPU computed elastograms at different window lengths	68
XII	Table of SNRe of CPU and GPU computed elastograms at different window lengths	68

LIST OF FIGURES

FIGURE		Page
1	Schematic demonstrating the principle of elastography (a) pre-compression (b) post-compression	8
2	Ideal strain profile of target shown in Fig.1	10
3	Time delay between pre- and post-compressed A lines	12
4	Computational cost of frequency domain versus time domain cross correlation with increasing number of sample points	16
5	Computational cost of time domain cross correlation with increasing applied strain percentage. Number of computations is plotted using C_t in Eq.2.11 with 5% window length	17
6	(a)0% stretch (b)1% stretch (c)2% stretch (d)3% stretch (e)4% stretch. Stretch factors affect the quality of the strain estimate . . .	19
7	(a)Axial strain map of non-homogenous target with 1 inclusion using gradient of displacements (b)staggered axial strain map using gradient of non-overlapping windows. (b) has better SNRe than (a)	24
8	(a)Axial strain map of non-homogeneous target with 2 inclusions using gradient of displacements (b)staggered axial strain map using gradient of non-overlapping windows. (b) has better SNRe than (a)	25
9	Strains at different window lengths as percentage of total acquisition depth (a)2.5% (b)3.75% (c)5% (d)6.25% (e)7.5%. Axial strain resolution deteriorates as window length increases	26
10	Strains at different window overlaps as percentage of total window length(a)60% (b)70% (c)80% (d)90% (e)95%. Axial strain resolution improves as window overlap increases	27

FIGURE Page

11 Percentage of time spent in different functions 28

12 Distribution of function calls in C implementation 29

13 Percentage of branch mispredictions in different functions 29

14 Percentage of L1-Instruction cache misses 30

15 Percentage of Data Transfer Look-aside Buffer load misses 31

16 Percentage of Data Transfer Look-aside Buffer store misses 31

17 System layout with the GPU connected via the PCIe bus 38

18 Low bandwidth between system memory and graphics memory limits performance gain from parallel execution on GPU 40

19 A Streaming Multiprocessor in GeForce 8800 GT 42

20 Schematic of GeForce 8800 GT card 44

21 Memory model in GPU computing 47

22 Percentage of CPU time spent on different functions during the execution of elastography implemented in C 51

23 Comparison of bandwidth obtained using paged memory versus pinned memory on 8800GT and E4500 Core2 Duo 57

24 Sequence flow in elastography (a) When both CPU and GPU are used for computation (b)When only GPU is used for computation . . 61

25 Time cost for generating axial strain elastograms using different versions of code 64

26 GFLOPS speed up achieved from executing functions in GPGPU . . 65

27 Graph of CNRe on CPU and GPU generated elastograms at (a)40dB SNRs (b) 20dB SNRs (c) 10dB SNRs 66

28 Graph of SNRe of CPU and GPU generated elastograms at (a)40dB SNRs (b)20dB SNRs (c)10dB SNRs 67

FIGURE Page

29 (a-c)GPU generated elastograms generated at 1mm, 2mm and 3mm window length. (d-f)CPU generated elastograms generated by the CPU algorithm at same values of window length 69

30 Images of a tissue mimicking phantom containing a stiff inclusion.(a)B mode sonogram (b)Elastogram on CPU (c)Elastogram on GPU 70

31 GPU time profile of strain estimation algorithm 71

32 Profile of uncoalesced loads from global memory during strain estimation on GPU 72

33 Profile of uncoalesced stores to global memory during strain estimation on GPU 73

34 Profile of divergent branching in strain estimation on GPU 74

35 Profile of warp serialization in GPU strain estimation 75

36 Occupancy profile in strain estimation on GPU 76

37 Ratio of CPU to GPU time for various functions 76

38 Profile of axial displacement estimation on GPU 77

39 Profile of sum of squares computation on GPU 77

40 Profile of Complex Multiplication on GPU 78

41 Profile of temporal stretching on GPU 78

42 Profile of median filtering on GPU 79

43 Profile of staggered strain estimation on GPU 79

CHAPTER I

INTRODUCTION

Elastography is an established medical imaging modality used to image the distribution of elastic properties of such as stiffness and elastic moduli as well as viscoelastic and poroelastic properties in a region of interest[1],[2],[3]. This technique maps the distribution of parameters related to the mechanical attributes in the target to false color coded visual information. In medical imaging, elastography is being studied for its potential as a diagnostic tool in detecting pathological changes in soft tissues [2]. While most of the processing is still performed offline, real-time elastography methods are quickly gaining popularity due to their potential high impact in clinical diagnosis [4],[5].

A. Motivation

Changes in tissue elasticity are related to the physiological health of the tissue. Changes in tissue stiffness may manifest as changes in tissue elasticity which may indicate pathogenic or malignant growth. A tumor is 5-28 times stiffer than the background of normal soft tissue [6]. For instance, scirrhous carcinoma appears as extremely hard nodules in the breast [1]. In standard medical practice, tissue elasticity is qualitatively assessed by palpation. While palpation may still be the preliminary diagnostic step, its subjectivity (perception of degree of stiffness may vary from physician to physician) make results more prone to inconsistency and hence less reliable.

The journal model is *IEEE Transactions on Automatic Control.*

Also, a small sized lesion or one that is embedded deep inside the tissue is hard to detect by palpation. This necessitates painful, invasive biopsies.

Among other medical imaging modalities, acoustic imaging techniques are most well suited for screening and routine diagnostic examinations of tissues that have strong sound contrast properties. Ultrasonic B-mode imaging has been used extensively in clinical applications ranging from obstetrics and gynecology to abdominal, cardiac and cancer imaging. Ultrasonic imaging works on the principle of acoustic reflectivity and regions with good contrast in echogenicity are detected well in the ultrasound image. However, two regions with the same echogenicity may have different stiffness contrast. For instance, tumors in the breast or prostate are much stiffer than the embedding tissue and scirrhosis of the liver increases the stiffness of the whole tissue, yet the tissues may appear normal in ultrasound scans [1]. Elasticity and echogenicity are uncorrelated and traditional B-mode imaging may not detect elastic contrast. Elastography can provide new information about areas opaque to sonography due to acoustic shadowing, areas with hard lesions in a soft background and isoechoic regions that are invisible to sonography.

This is the primary motivation behind elasticity imaging - to provide new information on tissue stiffness that can be complemented with echo contrast information available from ultrasound imaging in order to have a more clinically useful, specific and accurate diagnostic report. Much research effort has been directed toward its realization since its inception [7]. Though still a relatively novel technique in the area of imaging, elastography has evolved from a research bench to a diagnostic tool capable of providing information for improved diagnosis. Today, elastography is being considered as a potential replacement for painful biopsies.

B. Research Plan

The ultimate goal of elastography is to provide accurate information about the health of tissues that will enable detection of disease at real-time and aid in fast and objective diagnostic decision making. Elastography is based on estimation of displacement between a pair of rf signals which can be accomplished using cross-correlation techniques. Cross-correlation based estimators are accurate, robust and sensitive but are computationally intense. Their asymptotic performance of $O(n^2 n_w log_2(n_w))$, (where n is the size of the input data and n_w is the size of the correlation window) when operated on a pair of 2D rf data makes it challenging to employ them for real time processing on modest workstation hardware. Given the constraints of hardware, an algorithmic order that is hard to beat and image quality requirements that cannot be compromised, the key to tackling this speed-quality orthogonal problem may lie in identifying stages in the flow of the algorithm that are parallelizable and can be mapped efficiently to available cost effective parallel hardware.

1. Premise of This Thesis

The hypothesis in this work is that *cross-correlation based strain elastography can be parallelized and is well-suited for execution on parallel hardware providing real-time performance with no compromise in image quality*. This hypothesis is based on the knowledge that the most computationally intense stages of elastography algorithm are iterative with data-independent operations and are therefore parallelizable. General Purpose Graphics Processing Unit(GPGPU) engines provide efficient hardware platforms for parallel processing and have potential for massive performance lifts[8]. The engineering challenge involved in this work is in performing the mapping from sequential stages to GPGPU parallel hardware efficiently in order to maximize speed

and minimize loss of image quality.

2. Goals of This Research

The main goal of this work is to generate elastograms at real time frame rates by offloading computationally intensive stages to the GPGPU. The specific goals of this work are to

1. Develop software to generate axial and lateral strain elastograms
2. Identify data parallel stages in the algorithm
3. Offload these stages to GPGPU
4. Optimize execution on GPGPU by identifying opportunities and trade-offs
5. Verify for no loss in image quality
6. Compare performance of the CPU only and CPU-GPGPU versions
7. Identify trade-offs and scope of future work

3. Method and Preliminary Results

This study investigates the use of parallel general purpose graphics processing unit (GPGPU) engines for speeding up generation of elastograms at real-time frame rates while preserving elastographic image quality. To achieve this goal, a cross-correlation based time-delay estimation algorithm was developed in C programming language, executed on an Intel Core2 Duo Pentium machine and was profiled to locate performance blocks. The hotspots were addressed by employing software pipelining, read-ahead and eliminating redundant computations. The algorithm was then analyzed for parallelization on GPGPU and the stages that would map well to the GPGPU

hardware were chosen for execution on a GeForce 8800 GT GPGPU. Later, optimization principles of coalesced access, pinned memory, optimal thread occupancy were applied and a net improvement of 67x with respect to the original optimized C version of the estimator was achieved. For typical diagnostic depths of 3-4cm and elastographic processing parameters, this implementation can be predicted to yield elastographic frame rates in the order of 50fps. It was also observed that all of the stages in elastography have sub-optimal memory access patterns and also because data transfer from graphics card memory to system memory can be efficiently overlapped with concurrent CPU execution. Therefore a hybrid model of computation where computational load is optimally distributed between CPU and GPGPU was identified as an optimal approach to adequately tackle the speed-quality problem.

4. Impact and Significance of This Work

Abstraction of parallelism in the strain estimation algorithm and identifying that it conforms to the parallelism in the GPGPU hardware is the chief contribution of this work. Mapping of the parallel content of the software to the parallel hardware has to be efficient in order to exploit parallelism in hardware accelerators. GPGPU does not scale well for algorithms where this mapping is inefficient. For these cases, the cost of offloading work for GPGPU execution could outweigh the benefits. Careful analysis of acceleration opportunities like using fast on-chip shared memory, coalescing global memory access, minimizing non-unit strided access and warp serialization, maximizing occupancy, while recognizing the off-card memory bandwidth bottleneck are critical to achieving winning performance on the GPGPU platform. The results of this research corroborate the initial hypothesis that cross-correlation based elastography is parallelizable and that there is a significant improvement in performance to be had from efficient mapping of the software solution to a parallel hardware ar-

chitecture.

This work has led to positive changes in the way we work in our lab. Elastographic time constant imaging, bone imaging, ultrasound simulation have reported gaining over 48 hours of processing time using CPU-GPGPU implementation of strain elastogram as against the CPU-only implementation.

The results of this research suggest that use of GPGPU with CPU may allow generation of real time elastograms without significant compromise in image quality, a scenario that could be very favorable in real-time clinical applications. Availability of information about the mechanical state of tissues at real-time will aid in the early detection, precise diagnosis of disease in the tissue, ultimately reducing treatment cycle-time and hence costs spent by a patient.

5. Structure of The Thesis

This thesis is structured as follows. In Chapter II, we discuss the principle of elastography, the methodology and the implementation of the strain imaging algorithm in C. We also discuss the performance of CPU-only C implementation of these algorithms. In Chapter III, we discuss the features of GPGPU, the hardware configuration, the programming model and the trade-offs inherent in this hardware. We discuss the results of the first run of offloading the parallelizable stages to GPGPU. We investigate the standard optimization principles for exploiting the parallel hardware and discuss the results obtained by employing these principles. We also study the effects on image quality due to hardware limitations of the GPGPU, and the statistical differences in the image produced by execution on CPU and GPGPU. Chapter IV discusses the limitations of this implementation, options for enhancing the performance and suggests avenues for future work.

CHAPTER II

ELASTOGRAPHY ON CPU

A. Principle of Elastography

Elasticity imaging is typically done by processing the ultrasound RF data to estimate tissue displacements induced by external stimuli or internal motion. Quasi static compressions are used to excite the tissue externally in the direction of ultrasonic radiation [7],[9], [10], [2]. Alternatively, internal stimulus from inherent activity of the organs such as cardiovascular activity of the heart or blood flow can be used to produce elastographic signal [2]. The resultant speckle patterns contain information about internal displacement of the individual tissue components. Coherent echoes before and after compression, in the direction of applied strain, are then divided into overlapping windows in the time domain. The delay between these windows is tracked using speckle tracking methods such as cross-correlation [9]. Assuming the velocity of sound in the tissue is constant, the delay in time domain can be converted to longitudinal displacement between the adjacent windows. The resultant strain distribution is obtained by computing the gradient of displacement. The resultant strain images are referred to as 'axial strain elastograms'. Each pixel in an elastogram denotes the amount of strain $\hat{\epsilon}$ experienced by the tissue during compression, given by

$$\hat{\epsilon} = \frac{\hat{\tau}_1 - \hat{\tau}_2}{\Delta t} \tag{2.1}$$

where $\hat{\tau}_1$ and $\hat{\tau}_2$ denote the axial displacement estimates in windows 1 and 2 respectively separated by a distance of Δt.

Typically, $\Delta t \approx$ 0.1-0.2 mm [11]. The applied compression is typically in the range of 0.5-2% of tissue depth. The echoes are traced during or after the time that the tissue undergoes deformation caused by the excitation [12].

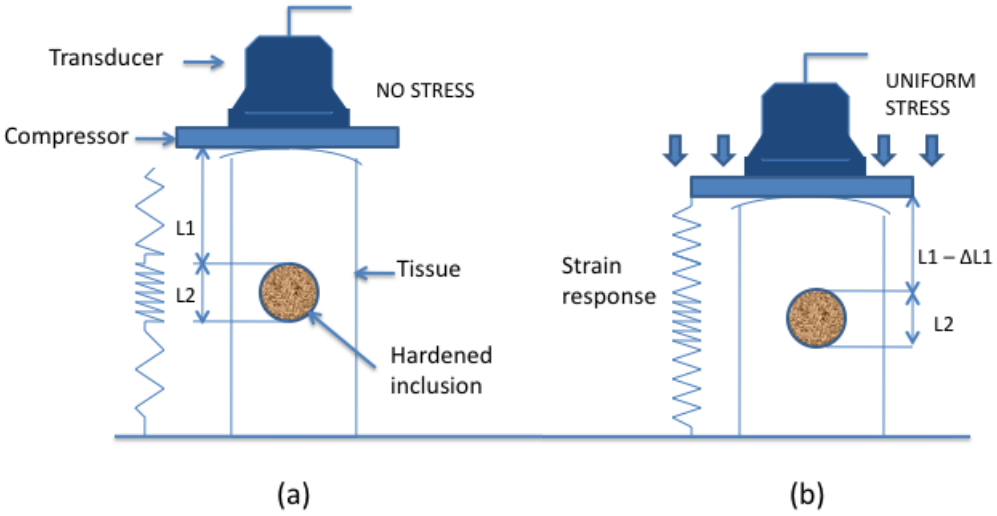

Fig. 1. Schematic demonstrating the principle of elastography (a) pre-compression (b) post-compression

A schematic of a typical elastography experiment is shown in Fig.1. The basic assumption made in tissue elastography is that the tissue behaves as an elastic, incompressible solid while in reality, it is viscoelastic. The assumption implies that there is a linear relationship between tissue stress and strain, the tissue is isotropic, there is no hysteresis, stress relaxation or creep. This assumption is justified in quasi-static elastography experiments by Ophir et al. [2].

In this simplistic model, the tissue is modelled as a cascaded spring with a rigid base as shown in Fig 1. This is the 1D spring model of a layered tissue. In Fig 1(a),

a transducer-compressor assembly is placed on the surface of the tissue, ultrasonic pulses are fired and the echo response of the uncommpressed tissue is recorded. In Fig.1(b), the tissue is uniformly compressed under quasi-static controlled conditions, and the echo response of the compressed tissue is recorded. In the case of uniaxial tensile stress in a cascaded spring assembly, the force in all the spring segments is the same. Consequently, the mathematical model of tissue is simplified and the equation of quasi-static uniaxial stress reduces to the Hookean equation [2]

$$F_o = K\Delta x \tag{2.2}$$

where

K : local stiffness of the tissue

F_o : Applied stress

Δx : resulting local change in displacement

The equivalent equation in this model for the continuous case becomes

$$Y = \frac{F_o}{\Delta x} \tag{2.3}$$

Plugging Eq.2.2 into Eq.2.3, we get

$$Y \propto K \tag{2.4}$$

From Eqn.2.4 we see that in the cascaded spring model, stiffness constant for a tissue region can be quantified by Young's Modulus [2]. Experiments have established that larger the area of the compressor, the more uniform the longitudinal stress fields and consequently more uniform strain fields [2]. Fig.2 shows the strain profile of the set up in Fig.1.

The level of applied strain is kept small to maintain the Hookean equation in the

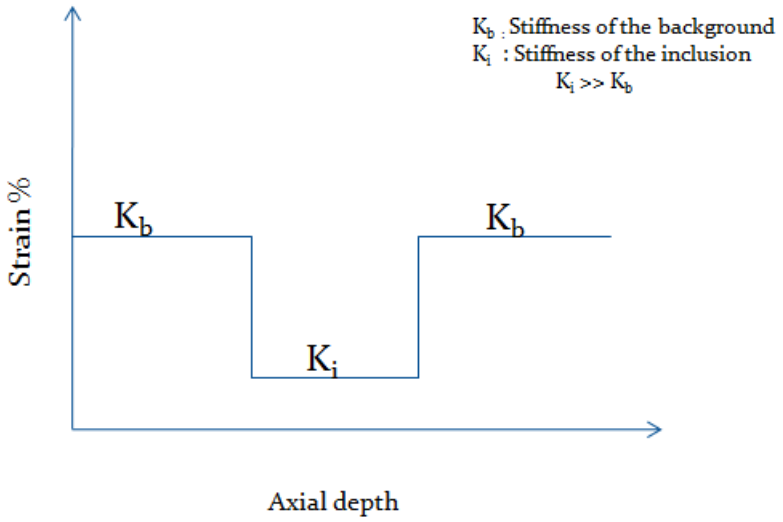

Fig. 2. Ideal strain profile of target shown in Fig.1

linear range of stress-strain relationship.

Strain is a 3D tensor, strain elastography is fundamentally a three dimensional problem with displacement in the axial, lateral and elevational directions. Though recent work on lateral and elevational strain estimation by Lubinski et al. [13], Chaturvedi et al. [14], [15] and Konofagou and Ophir [16] suggest that it is possible to generate lateral and elevational elastograms, in this study we will focus only on axial displacement and axial strain estimation. The concepts and approaches developed in this work, however, can be easily extended to lateral and elevational strain elastography.

B. Methodology

The steps to generate axial strain elastograms are described in the following sections:

1. Acquisition of Tissue Motion Data

A set of digitized RF echo data is obtained after placing an array ultrasound transducer on the surface of the target tissue. An array allows acquisition of several A lines simultaneously by coherent excitation of array elements. The scanner usually operates between 1MHz and 20MHz in order to optimize for resolution and penetration [17]. The surface is then slightly compressed with the transducer or with a transducer-compressor assembly, and another set of digitized and compressed RF echo data is obtained from the same area of interest. The pre- and post-compression signals are independently stationary but jointly non-stationary and this should be taken into account while processing these signals.

2. Time Delay Estimation for Axial Displacement

A time delay between the pre- and post-compressed echo signals arise from the spatial shift of the compressed tissue. Assuming the speed of sound in the soft tissue is constant, the spatial shift is proportional to the time shift. Hence, delay estimation in time domain is equivalent to displacement estimation in spatial domain. Fig.3 shows the time delay between pre- and post-compressed A lines in elastography.

The quality of elastograms depends directly on the ability to estimate time delay accurately [9]. The presence of noise in the post-compressed echo signal induced by mechanical compression, imposes a limit on the accuracy achievable in time delay estimation [9], [2], [18], [17].

Time-delay estimation(TDE) can be performed using several methods [19],[20]. Available estimators are Sum of Absolute Difference (SAD), Minimum Mean Square Error(MMSE), Cross-Correlation based tracking algorithms[7], Fourier-based phase-tracking techniques [21] etc. Cross-correlation techniques are optimal for quasi-static

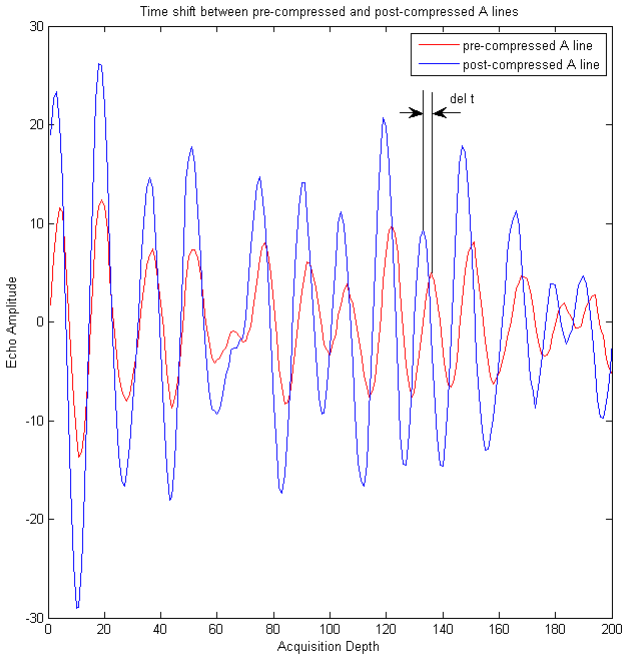

Fig. 3. Time delay between pre- and post-compressed A lines

applications [9], [2] and are hence preferred over other methods when precision is a priority.

In our implementation, local displacements are estimated by measuring time shifts in short time histories. The resultant displacement between the gated pre-compression and post-compression echo signal segments is estimated as the location of the peak of cross-correlation between the pre- and post-compression signals in that window of observation. Given the expression of the cross-correlation as:

$$\hat{r}_{xy}[\tau] = Ex[n]y[n+\tau] = \frac{1}{N}\sum_{i=1}^{N}(y[i]x[i+\tau]) \tag{2.5}$$

where

x: pre-compressed signal

y: post-compressed signal

$\hat{r}_{xy}$: estimated cross-correlation between pre- and post-compression signals

τ: time delay at which cross-correlation between between pre- and post-compression is maximum

N: the number of sample points in a window

The cross correlation window is translated for all depths. Each window of observation is shifted by a pre-defined linear distance till the last window of observation is reached for all depths of observation.

A more precise estimator of the time shift is the normalized cross-correlation function or the cross-correlation coefficient function(CCF) [22], given by

$$\hat{\rho}_{xy}[\tau] = \frac{\hat{r}_{xy}[\tau]}{(\hat{r}_{xx}[0](\hat{r}_{yy}[0])} \tag{2.6}$$

where

$\hat{\rho}_{xy}$: estimated cross-correlation coefficient between pre- and post-compressed signals

$\hat{\rho}_{xx}$: estimated auto-correlation of the pre-compressed signal

$\hat{\rho}_{yy}$: estimated auto-correlation of the post-compressed signal

τ: time delay at which cross-correlation between between pre- and post-compression is maximum

The time shift error in the normalized cross-correlation function is significantly lower than in the unnormalized cross-correlation function and is hence preferred over the latter [9].

a. Computational Cost of Time Delay Estimation

If cross correlation coefficient function is implemented in the frequency domain, and benchmark FFT routines are used, the data segments corresponding to the window of observation should be extended to have a length that is a power of 2. If the length of each window is N_w, it is zero padded till the length of this segment is N where N is the smallest power of 2 bigger than N_w. FFTs are computed for these data segments of the pre-compression and post compression pairs. The product of the Fourier transforms of the two data segments are Inverse Fourier transformed, and normalized to return the cross correlation coefficient. The total computational cost associated with these operations is given by

$$C_f = 3 * C_{FFT} = (3 * N \ln N) > (3 * N_w \ln N_w) \tag{2.7}$$

Instead, if cross-correlation is implemented in the time domain, we can exploit the redundancy inherent in elastography imaging [9]. The effective displacement in the entire depth of the tissue cannot exceed the applied displacement. If N_s is the sample equivalent of the applied strain, within a data window N_w, net displacement

cannot exceed N_s. Therefore, cross correlation needs to be computed for lags only until the time sample equivalent of the applied displacement, and not for the entire length of the data segment. For instance, if 2% strain is applied, and $N_s = 2\%N_d$, then cross correlation between pre-compressed and post compressed RF data needs to be computed only for N_s sample lags and not beyond. The peak of cross-correlation will be present in lags in the range of $[0 - N_s]$. The computational cost in time domain cross-correlation coefficient implementation for each window N_w is then given by

$$C_t = N_s * N_w \tag{2.8}$$

N_w is chosen such that $N_w \geq N_s$. If we assume D to be the total depth of the tissue, we have the following relation

$$N_s = s\%ofD \tag{2.9}$$

and

$$N_w = w\%ofD \tag{2.10}$$

Then, Eq.2.8 for the time domain cross-correlation becomes

$$C_t = (s\%ofD) * (w\%ofD) = \frac{s}{100} * \frac{w}{100} * D^2 \tag{2.11}$$

On the other hand, Eq.2.7 for the frequency domain cross-correlation becomes

$$C_f = 3 * (w\%ofD) \ln(w\%ofD) = 3 * (\frac{wD}{100}) \ln(\frac{wD}{100}) \tag{2.12}$$

Finally,

$$\frac{C_f}{C_t} = \frac{3 * (\frac{w}{100} * D) \ln(\frac{wD}{100})}{\frac{s}{100} * \frac{w}{100} * D^2} = \frac{3 * \ln(\frac{wD}{100})}{\frac{s}{100} * D} \tag{2.13}$$

Eq.2.13 would have interesting implications for cross-correlation based elastography applications. Just as a proof of principle, if we assume a tissue of 40mm depth,

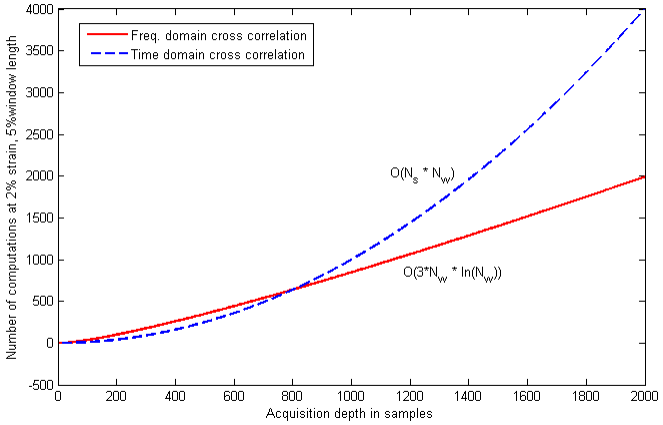

Fig. 4. Computational cost of frequency domain versus time domain cross correlation with increasing number of sample points

compressed by 2% and use a window length of 2mm which corresponds to 5% of total depth D, we have $w = 5$ and $s = 2$. Plugging these values into Eq.2.13 we get,

$$\frac{C_f}{C_t} = \frac{3 * \ln(.05D)}{.02D} \tag{2.14}$$

Asymptotically, C_t will grow faster than C_f. As can be seen from the plot in Fig.4, at smaller values of depth D ($\leq$ 2000 sample points), we get $\approx$ 2-3 times gain in execution time on using time domain cross-correlation. However as D increases, the rate of growth of C_t outweighs that of C_f.

In time domain, as the percentage of applied strain increases, the number of computations increase exponentially as can be seen in the plot in Fig.5. On the other hand, computational cost of the frequency domain implementation is independent of the applied strain.

In terms of computational efficiency, time domain cross correlation better than

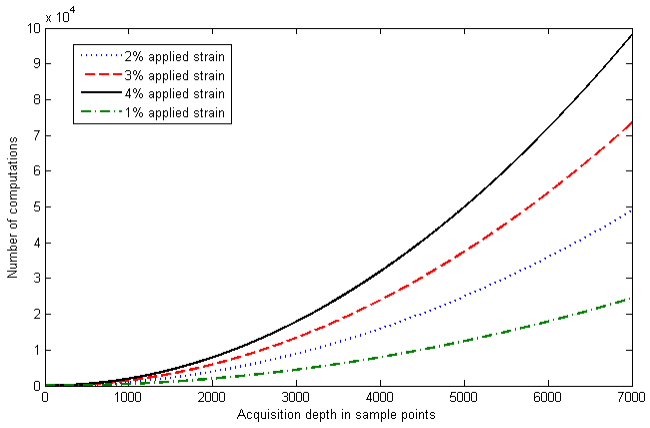

Fig. 5. Computational cost of time domain cross correlation with increasing applied strain percentage. Number of computations is plotted using C_t in Eq.2.11 with 5% window length

frequency domain cross correlation when the number of sample points is small($\approx$ 2000). For typical mechanical parameters of strain, tissue depth and signal processing parameter of window length, time domain implementation may outperform frequency domain implementation by 2-3 times [9]. If the applied strain is more than 2%, and the number of samples is large ($\approx$2000 and above), FFT based implementation soon outperforms time domain implementation in both execution time and image quality. With high N, time based implementation has an asymptotic growth of N^2 while frequency based implementation grows at 3Nln(N). Time domain implementation gains over frequency based implementation only for small strains (typically $\leq$1%) and small input samples.

In our implementation, we have used FFTW libraries version 2.x for computing discrete fourier transform in O(NlnN) time. The FFTW package, developed by Matteo Frigo and Steven G. Johnson at MIT, provides portable, scalable and one of the

fastest implementations of FFT in C.

3. Signal Conditioning

The quality of time delay estimation depends on the extent of similarities between the pre-compressed and post-compressed echoes. In elastography, the amount of similarity is reduced due to the parameters involved in data acquisition. When mechanically compressed, the tissue scatterer spacing is reduced and the resultant echoes reflected from physically compressed acoustic scatterers will be distorted [9], [2]. Note that this distortion also constitutes the actual strain that is displayed in the elastograms. Due to this distortion, cross-correlation between an uncompressed echo and another temporally compressed echo will be poor since the compressed echo is no longer a delayed replica of the uncompressed echo. This is referred to as decorrelation noise [2].

To partially correct this, the post compressed echo is usually temporally stretched prior to CCF computation. This step can almost entirely clean the signal of decorrelation noise and improve the SNR dramatically [23]. Essentially, the stretching 'realigns' the scatterers within the correlation window. An appropriate stretching factor will make the post compression echo a closer replica of the pre-compression echo and the cross correlation will improve considerably. The choice of stretching factor is based on the amount of applied strain. This is a constraint of this method - apriori knowledge of the applied strain is required to influence effective denoising. Other methods logarithmic amplitude compression and soft-limiting do not require prior knowledge of the applied strain [9].

Temporal stretching is usually implemented using linear interpolation. Though linear interpolation is not the most accurate way to effect temporal stretching [9], we used this method because it is simple, fast and also improves accuracy with oversam-

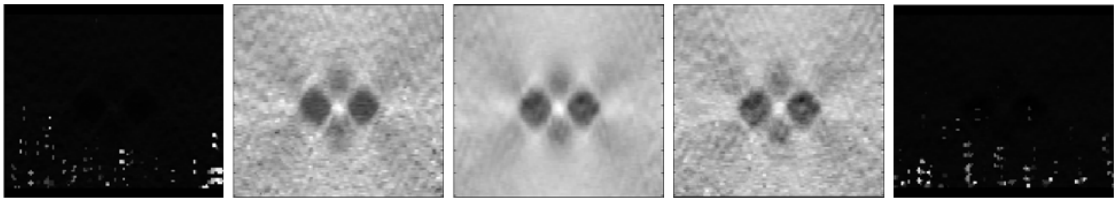

Fig. 6. (a)0% stretch (b)1% stretch (c)2% stretch (d)3% stretch (e)4% stretch. Stretch factors affect the quality of the strain estimate

pling. Stretching can be done either globally, where all windows are stretched equally, or adaptively, where windows are stretched by different factors. Adaptive stretching is iterative and computationally intensive. For small strains typical of real-time applications, global stretching is usually acceptable for signal conditioning. However, it is important to note that global stretching works optimally when the target is homogeneous. When the target is non-homogeneous, stretching the target globally with a factor in the order of the applied strain would imply over-stretching low-strain areas inside the inclusion or under-stretching high strain areas in the background. This has the potential to corrupt the strain image. Hence a global stretch factor should be chosen carefully. Fig.6 shows a practical simulation example to illustrate the effect of stretching on non-homogeneous targets.

a. Computational Cost of Signal Conditioning

Each A line is extended by the stretching factor and the echo amplitudes are computed at the new points. In each A line, there are N new points at which echo amplitudes have to be recomputed. Computational cost of temporal stretching is therefore, O(N^2).

4. Interpolation

In the cross-correlation approach, the time delay obtained between the pre-compressed and the post-compressed A lines is an integral multiple of the pixel sample interval. This is the time quantization error due to digitization of RF data. In elastography where the applied strain is in the range of 0.5%-2%, the actual time delay is much smaller than the sample interval. To estimate sub-sample displacement values, it is required to interpolate between samples [9]. One method to perform interpolation is by oversampling. However, this method is computationally inefficient since it increases the length of the entire cross-correlation function while we need finer resolution only near the peak [9], and hence is not used here. Two efficient interpolators used in this implementation are cosine interpolator and parabolic interpolator.

a. Parabolic Interpolation

This is a polynomial interpolation method that uses 3 points - the estimated peak and its left and right neighbor points to compute the quadratic order polynomial passing through them. If t_1 is the position of the estimated peak y_1 of the cross correlation function and y_0 and y_2 are the left and right neighbors of y_1 respectively, the interpolation function around these 3 points is given by [9]

$$y(t) = y_0\frac{(t-t_1)(t-t_2)}{(t_0-t_1)(t_0-t_2)} + y_1\frac{(t-t_0)(t-t_2)}{(t_1-t_0)(t_1-t_2)} + y_2\frac{(t-t_0)(t-t_1)}{(t_2-t_0)(t_2-t_1)} \tag{2.15}$$

Allowing $\Delta T = t_1 - t_0 = t_2 - t_1$, Eq 2.15 reduces to the form

$$y(t) = at^2 + bt + c \tag{2.16}$$

where

$a = \frac{(y_0-2y_1+y_2)}{2\Delta T^2}$

$b = -\frac{(y_0(t_1+t_2)-2y_1(t_0+t_2)+y_2(t_0+t_1))}{2\Delta T^2}$

$c = \frac{y_0(t_1)(t_2)-2_{y1}(t_0)(t_2)+y_2(t_0)(t_1)}{2\Delta T^2}$

The maximum of Eq 2.16 is at $t_p = \frac{-b}{2a}$. The distance of the true peak from the estimated peak is $\delta = t_1 - t_p$. The estimate of this distance is given by

$$\hat{\delta} = \frac{(y_2 - y_0)}{2(2y_1 - y_2 - y_0)}\Delta T \tag{2.17}$$

Parabolic interpolation is a biased estimator of the true location of the peak, the bias being a function of the fractional part of the true location of the peak[9]. The bias error is minimum when the estimated peak coincides with the true peak, or the true peak is half-way between the two samples. The bias error is maximum when the true peak is about $.25\Delta T$ distance from the estimated peak. In our work, we use the parabolic interpolator to detect any lateral motion while estimating axial displacement.

b. Cosine Interpolation

This is a trigonometric interpolation method that fits a cosinusoid to the largest 3 samples of the cross-correlation function. If y_1 is the peak of the cross correlation coefficient function at t_1 and y_0 and y_2 are its left and right neighbors at t_0 and t_2 respectively, the angular frequency of and phase of the cosine function are given by Eq. 2.18 and Eq. 2.19 respectively

$$\omega_0 = cos^{-1}(\frac{y_0 + y_2}{2y_1}) \tag{2.18}$$

$$\theta_0 = tan^{-1}(\frac{y_0 - y_2}{2y_1 \sin\omega_0}) \tag{2.19}$$

The interpolated peak is at $\hat{\delta}$ given by

$$\hat{\delta} = \frac{\theta_0}{\omega_0} \tag{2.20}$$

The new peak is at $t_p = t_1$ - $\hat{\delta}$ and the value of the interpolated peak is $\frac{y_1}{\cos\hat{\delta}}$. Like the parabolic interpolator, the cosine interpolator is a biased estimator and the bias is minimum when the true peak coincides with the estimated peak, or is half-distance off the estimated peak, and is maximum at about when the true peak is about .25 distance off from the estimated peak. Cosine interpolation is used for enhancing precision of axial displacement estimates.

c. Computational Cost of Interpolation

Each peak in a lateral kernel of search is interpolated using the parabolic interpolator and this is repeated for every axial window. The computational cost of parabolic interpolation is order of $O(N^2)$. Each peak in an A line is cosine-interpolated to detect any sub-sample peak. This is repeated for every A line. The order of this interpolation is therefore, $O(N^2)$.

5. Median Filtering

Median filtering is an image engineering technique of noise-smoothing. Its edge preserving feature makes it more useful than low-frequency linear filters in medical imaging applications. It is highly effective for smoothing salt-pepper noise. Median filtering is also computationally more accurate, because it relies on numerical comparisons and is not prone to overflow or rounding errors which may occur in linear filtering implementations[24].

a. Computational Cost of Median Filtering

In our implementation, the median of each neighborhood is computed using the quicksort C library whose complexity is O(Nln(N)), where N is the number of elements to be sorted. If the size of the filter window is N_wxN_w, the computational cost of median filtering is O($N^{2*}(N_w^2 \ln(N_w^2))$). Much more efficient implementations of O(N_w^2), O(log(N_w^2)) [25] of computation of median have been proposed. Using any of these implementations is suggested as future work.

6. Strain Estimation

Axial strain is the spatial derivative of the displacement along the insonification axis. It is estimated by computing the local gradient of displacement over adjacent overlapping windows. High window overlaps generate more pixels in the elastogram but it also introduces large noise. This degrades the SNRe of the strain estimate, rendering it less than useful for detailed diagnosis. To overcome this problem, a multi-step strain estimation technique is employed. This technique has been referred to as staggered strain estimator[26]. In the first iteration, gradient is computed for non-overlapping adjacent windows. In the subsequent iterations, data windows are shifted by a fraction of the window length and strain is estimated for non-overlapping windows with this window translation. This is repeated till the cumulative window shift equals or exceeds the window length. At this point, the elastograms from all the iterations are superimposed on a final image, each image staggered by their window shifts. Staggered strain estimation improves the CNRe and SNRe significantly without adversely affecting the spatial resolution. Fig.7 and Fig.8 illustrate the difference between strain image generated by gradient method and one generated by staggered strain method for non-homogeneous targets with 1 inclusion and 2 inclusions respectively.

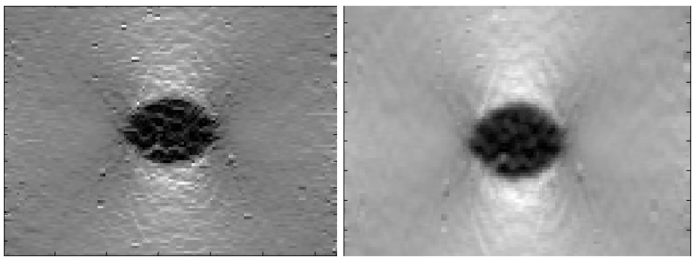

Fig. 7. (a)Axial strain map of non-homogenous target with 1 inclusion using gradient of displacements (b)staggered axial strain map using gradient of non-overlapping windows. (b) has better SNRe than (a)

a. Computational Cost of Strain Estimation

In axial strain estimation, for each point in an A line, the computation done is

$$\Delta S_i = \frac{d_i - d_{i-step}}{StepSize} \tag{2.21}$$

The order of this computation for the entire image is O(N^2).

C. Image Quality Analysis

Performance of elastography is controlled by 3 groups of factors [10],[27]:

1. Ultrasonic parameters : Transducer center frequency, fc, Bandwidth B, sonographic SNR and beam width, pitch, sampling frequency [2]

2. DSP parameters: Length of the cross correlation window w, shift between two consecutive cross correlation windows, Δw [2], [17]

3. Mechanical parameters : True elastic modulus, effective Poisson's ratio and boundary conditions[28] and [2]

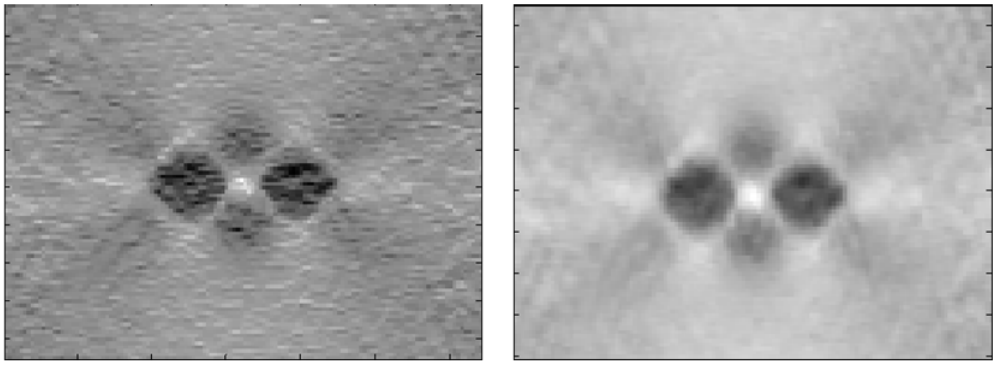

Fig. 8. (a)Axial strain map of non-homogeneous target with 2 inclusions using gradient of displacements (b)staggered axial strain map using gradient of non-overlapping windows. (b) has better SNRe than (a)

These three sets of parameters are somehow interdependent and need to be in agreement with each other for optimal performance. For instance, the window length, a DSP parameter, needs to be a function of the ultrasonic wavelength, an acoustic parameter. Any change in the input parameters should be accompanied by adjusting the interdependent parameters. When the applied strain is increased, the stretch factor by which to reduce the decorrelation noise has to be increased in order to retain the quality of the elastograms.

For the purpose of illustration, Fig.9 shows how the resolution of strain images is affected by the length of the cross-correlation windows. At smaller window lengths, the SNR of the elastograms is low while the resolution is good. As the size of the window length increases, the SNR improves while the resolution deteriorates. Window overlap is another signal processing parameter that affects the quality of elastograms [2], [17]. Fig.10 shows how the resolution of a non-homogeneous target with 2 in-

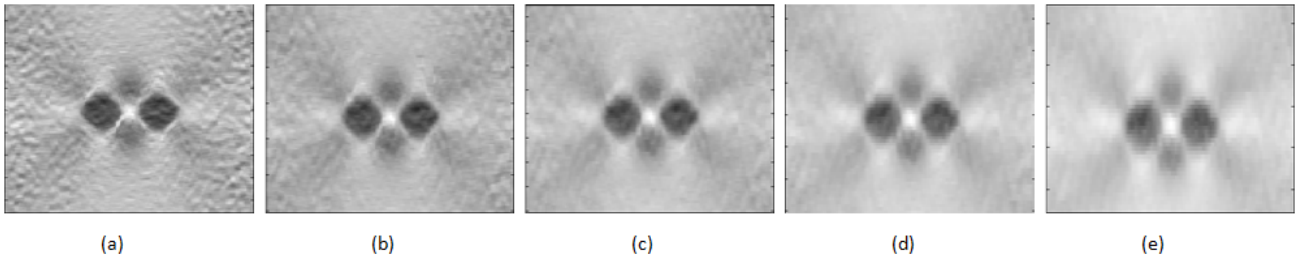

Fig. 9. Strains at different window lengths as percentage of total acquisition depth (a)2.5% (b)3.75% (c)5% (d)6.25% (e)7.5%. Axial strain resolution deteriorates as window length increases

clusions improves with increasing window overlaps. However, the upper bound of achievable resolution is defined by the bandwidth of the transducer [17].

D. Performance Analysis

In this section, we analyze the performance of the software implementation of elastography algorithms in C. For the purpose of analysis, we consider an axial strain elastogram of resolution 95x128 elements generated from a pair of 5195x128 element pre-compressed and post-compressed simulated RF data. The simulated RF data are generated at 6.5MHz. A window size of 5% of the total axial depth is used, and a window shift of 20% is applied. Visual Studio 8.0 development environment was used for implementation. The hardware configuration used for profiling is specified in Table I. Intel's VTune performance analyzer is used for profiling CPU time, branch mispredictions and cache performance.

From the time profile graph in Fig. 11, we can see that the process spends maximum time ≈50% in computation of cross-correlation. The function that searches

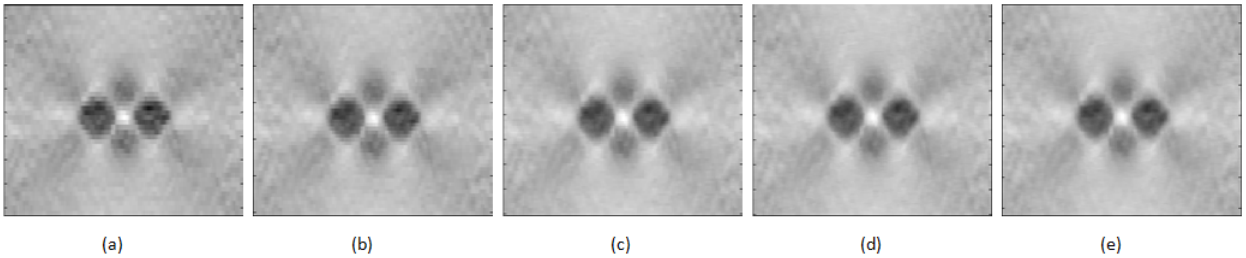

Fig. 10. Strains at different window overlaps as percentage of total window length(a)60% (b)70% (c)80% (d)90% (e)95%. Axial strain resolution improves as window overlap increases

Table I. Hardware configuration of E4500 Intel Core2 Duo

E4500 Intel Core2 Duo Specifications	
Core frequency	2.2GHz
FSB frequency	200MHz
Number of cores	2
Total amount of RAM	2GB
Total amount of L1 cache	64KB
Total amount of L2 cache	2MB

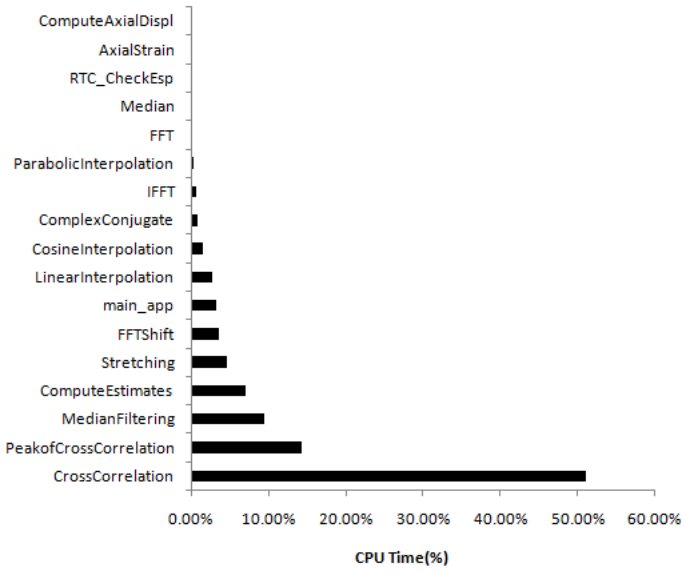

Fig. 11. Percentage of time spent in different functions

for the maximum peak consumes ≈15% of CPU time. Median Filtering is the third biggest cpu cycle consuming function at ≈10% of the CPU time. Linear, cosine and parabolic interpolation together consume ≈4.5% of the time while stretching consumes another 5%. Each run of FFT and IFFT using FFTW libraries accounts for only about .8% of the time while each fftshift consumes ≈3.5% and conjugate computation costs ≈.77%. Axial strain computation accounts for a relatively negligible .1% of CPU time. The rest of the time is attributed to function call, initialization and cleaning up overhead. Fig.12 shows the distribution of function calls in our implementation of elastography in C.

Branch mispredictions hurt pipeline performance and CPU cycles are wasted in flushing the pipeline instead of doing useful work. From Fig. 13, we can see that median filtering accounts for maximum percentage ≈43% of branch mispredictions.

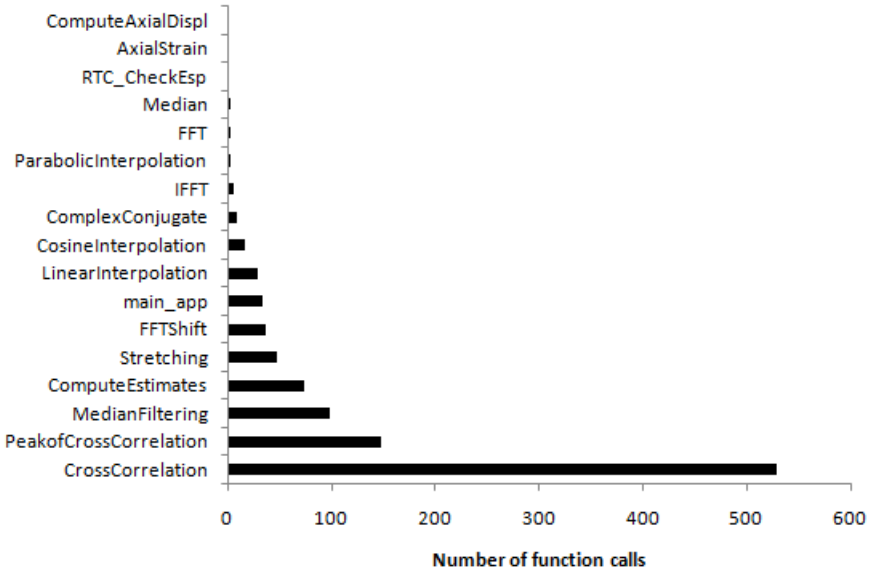

Fig. 12. Distribution of function calls in C implementation

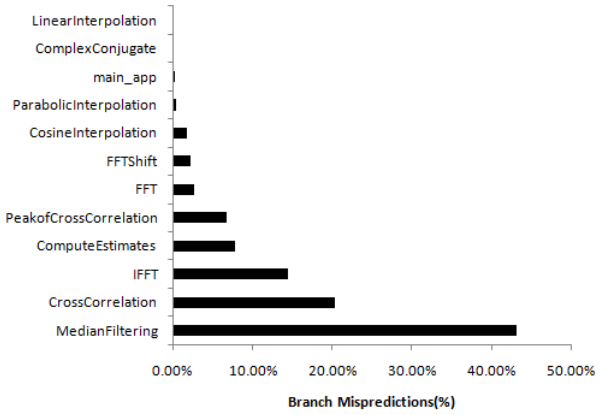

Fig. 13. Percentage of branch mispredictions in different functions

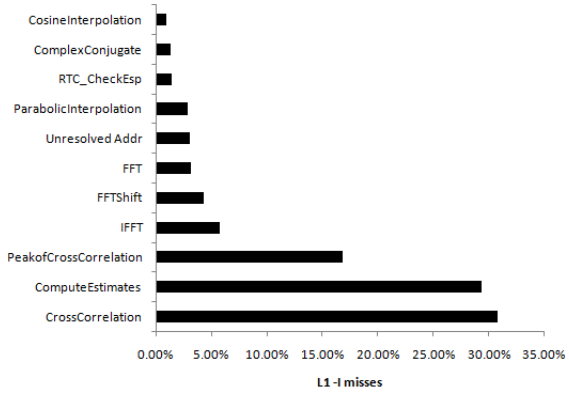

Fig. 14. Percentage of L1-Instruction cache misses

The high branching density in median filtering is due to window-based processing of each pixel, and also due to the sorting involved in computing the median. In our implementation, we have used the quicksortK algorithm. A more efficient sorting technique will mitigate the costs incurred due to branch misprediction. FFT and IFFT modules together account for about 17% of misprediction costs.

Finally, cache misses add to memory access latency and limit overall performance. Fig. 14 shows the distribution of misses in L1 Instruction cache among various functions. Cross-Correlation and computation of its peak together account for ≈47% of cache misses while FFT and IFFT suffer ≈8% of cache misses.

DTLB misses in load-store are the misses in the Translation Look Aside buffer which is the hardware cache for address maps between virtual memory and physical memory. Stretching module has significant DTLB misses during both load and store as seen in Fig.15 and 16.

The first version of the software generated elastograms at ≈ .142fps. By identifying repeated computations that were loop invariant, and performing these compu-

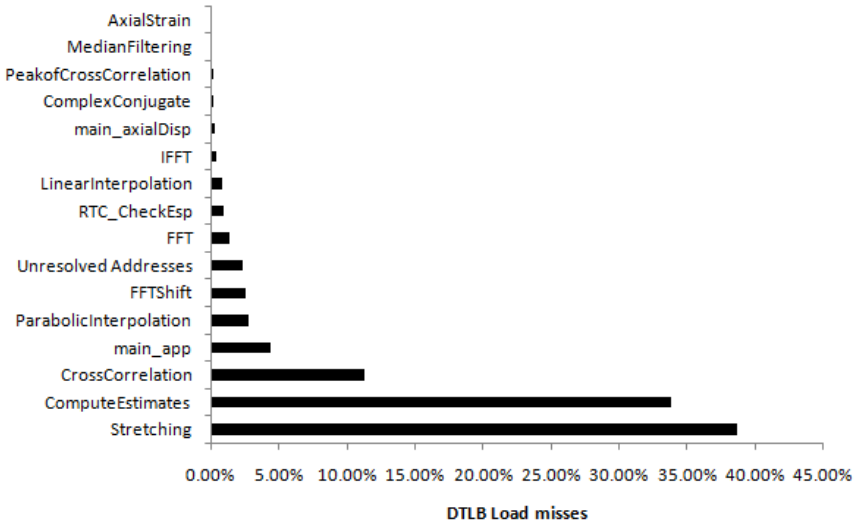

Fig. 15. Percentage of Data Transfer Look-aside Buffer load misses

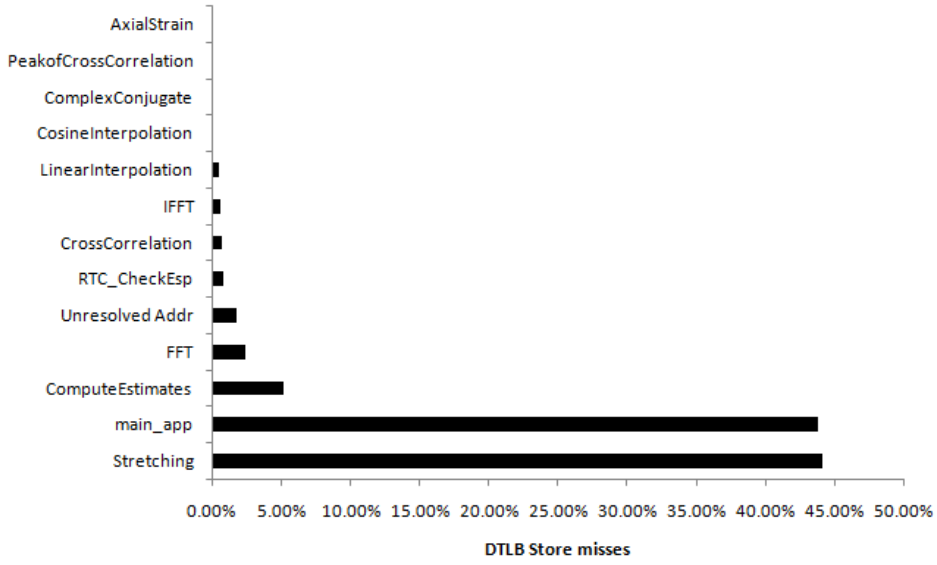

Fig. 16. Percentage of Data Transfer Look-aside Buffer store misses

tations outside the loop, we were able to generate elastograms at .285fps, an improvement of ≈2x. As can be seen from the profile reports in Fig.11-16, there is scope for improvement in cache and branching performance. The CPU usage of the application is pegged at ≈50%. Improving CPU utilization by employing additional optimization principles of software pipelining, out-of-order execution, loop unrolling etc. may give us better performance. Finally, the current implementation is single-threaded. Modifying this implementation to have a multi-threaded version that balances computational load between the two cores in the given platform can be estimated to yield a performance lift of an additional 2x. However, given the constraints of time and resource, we decided not to invest on this additional programming effort.

CHAPTER III

PARALLEL ELASTOGRAPHY ON CPU-GPGPU

A. Parallel Computing on GPGPU

The hunger for computational horsepower is never-ending. Powerful cores crunching 2 billion operations per second could not contain the desire for more power for too long. The multi-core hyperthreaded systems at the turn of the 21st century only increased our appetite for more power. The gaming industry was one of the first to respond by belting out high resolution 2D interactive games at real time speeds. It did so by employing a dedicated graphics engine with many more times ALUs to take care of render related volume based processing such as geometry and position of vertices, coloring of pixels that do not require separate instructions for a range of data points, while leaving flow control with the CPU. This division of labor ensured that precious CPU cycles are not wasted executing the same instruction on multiple data points, but are saved for crucial control required in interactive gaming, while the graphics engine took care of enriching visual experience.

This programming model earned huge acknowledgement from gaming enthusiasts and computational scientists alike. High performance computing groups involved in computational biology, physics simulation, astronomy saw the potential in this model to suggest a paradigm shift in computing, without having to pump up CPU clock speeds which was anyway showing signs of diminishing returns having hit the power wall. By having more transistors dedicated for performing arithmetic and logic operations than for performing control operations, GPUs were the new auxiliary

compute devices. CPUs wielding control, would be the master and GPUs driving the ALUs to perform the computations, would be the slave. PC based 3D graphics were raised to a new level of visual experience. In 2006, NVIDIA made public its Single Instruction Multiple Data programming model based CUDA (Compute Unified Device Architecture) suite with libraries and programming interfaces to program the graphics processor unit to perform non-graphics computation [29]. In the meantime, ATI and AMD also offered developers their computing language CTM(Close to Metal) and CAL(Compute Abstraction Layer) to program their GPUs for general purpose computations. The GPGPU had arrived. What was used for gaming would now be used for meeting medical imaging, space exploration, weather forecasting, financial market analysis and other numerical, scientific and engineering challenges [8].

The hardware architecture of the graphics engine has undergone significant change - logic blocks wired for specific graphic functionality as vertex shading or pixel shading or texture shading are now programmed to perform general computation. While GPGPU has become one of the most sought after research areas in the academia and research industry, the entertainment industry still remains the driving force for ever-increasing benchmarks in this domain, both in terms of funds and end-user expectations.

Recent advancements in 3D gaming has pushed the envelope even further. GPUs are now designed for anti-aliasing, high dynamic range, life-like shadowing making animation get incredibly closer to realism. With GeForce 8800 GTX, Games are available at Extreme High Definition resolutions of 2560x1600, at a clarity 7 times more than 1080i HD [29]. More and more ALUs per chip are making the FLOPS(floating point operations per second) reach new peaks. The GPU and system memory bandwidth has expanded more than 16x from PCI to PCIe in the last 3 years. The number of programmable computational shader units per Nvidia GPU card varies from 8 as

is in 8400 GS to 480 as is in GTX 295 [29].

Increased programmability also makes GPU a better tool for science and engineering applications. Programming libraries with simple C-like interfaces bring massive parallel programming to the reach of students and developers on desktop workstations. Compute Unified Device Architecture (Nvidia), Compute Abstraction Layer(ATI), Brook+(AMD) are user-friendly programming languages that enable application software to transparently scale its parallelism to leverage the increasing number of processor cores. A compiled CUDA program can execute on any number of processor cores because the runtime system will schedule the parallel execution while the programmer concentrates on the functionality. [30].

The tremendous computational power of GPGPU is due to its massively parallel architecture and is best harnessed by Single-Instruction-Multiple-Data(SIMD) algorithms. SIMD algorithms map very well to the parallel organization of the GPGPU computational units. When a SIMD kernel is launched on the GPU, the same set of instructions is dispatched to all active multiprocessors each of which can process independent chunks of data in parallel. For best performance, it is crucial for the input data elements to be independent. If there is data-dependence, the execution falls back to a sequential model and performance will be limited.

Algorithms that are most well suited to this constraint are data parallel operations such as filtering, scaling and transformations. These algorithms involve little or no dependency between data elements and hence processing on them can be in parallel. Most medical imaging, video processing, simulation problems have significant portions of code performing such data parallel operations and are hence popular test-beds for benchmarking. Even algorithms that are not entirely explicity SIMD or MIMD can be tuned such that the data-parallel sections of the code execute on GPGPU while the sequential code executes on CPU to register positive speed-up.

Future GPGPUs will be adapted well to accelerate task-parallel algorithms[31].

1. Elastography on GPU

Elastography is a non-invasive imaging modality that maps local strains experienced by soft tissues to color coded information. Elastography shows promise as a cost-effective tool in the early detection of diseases in soft tissues [7]. Conventional elastography algorithms use cross-correlation techniques for estimation of elastic displacements. Cross-correlation is a precise estimator but the estimation process is computationally intensive and not suitable for real time applications. Other available real time elastography applications usually employ less computationally intensive displacement estimation methods. These include sum of square difference, spectral strain estimation [20], phase root seeking [32] and zero-crossing track [33]. However, improvement in computational speed is often accompanied by losses in the quality of the resulting images.

Given the numerous instances of speed ups in various fields from computational modelling to astronomy [8], [29] and given our knowledge of data-parallel stages in elastography techniques, our hypothesis is that it is possible to map our previous implementation of ultrasoound elastography to this programming model and achieve real-time performance with no loss of image quality. In this study, we demonstrate how we tackle the speed-quality orthogonal problem by identifying parallelism in the cross-correlation based strain estimator and exploiting parallelism in GPGPU to display elastograms at real time frame rates with no loss in image quality.

We also note that not all of the stages in elastography should be offloaded to GPGPU because some stages have sub-optimal memory access patterns and also because data transfer from graphics card memory to system memory can be efficiently overlapped with concurrent CPU execution. Therefore, a hybrid model of computa-

tion where CPU and GPGPU work as peers is expected to adequately solve the speed challenge.

The rest of this chapter is organized as follows: discussion of the hardware configuration used in this research, the programming model used to parallelize the elastography software, base-GPU version and optimized version of the software, speed up of the CPU-GPU version with respect to the CPU-only version and scope for future performance improvement.

B. Scientific Computing with GPGPU

To harness the potential for acceleration in GPGPU, it is important to understand the features and limitations of this parallel computing paradigm. In the following sections, we briefly review the hardware features, the CPU-GPU communication bandwidth and latency limits, the memory model, the programming model and the execution model.

In a CPU-GPGPU programming scheme, the CPU with powerful control and coordination mechanisms assumes the role of a the master device and the GPGPU with hundreds of cores that run lightweight computational threads in parallel is the slave device. The CPU issues data-parallel commands and the GPGPU executes them. When the CPU and GPGPU assume a master-slave role as described above, the CPU is referred to as the 'host' and the GPGPU is as the 'device'[29].

In this study, the hardware configuration used is:

1. Intel Core2 Duo E4500 system with 2.2GHz CPU, 2GB system memory
2. Nvidia GeForce 8800 GT card with G92 core GPGPU, 500MB on-card memory

A discrete 8800 GT graphics card is connected to the motherboard via the PCIe slot. The graphics card hosts the GPU and a memory module. The GPU is connected

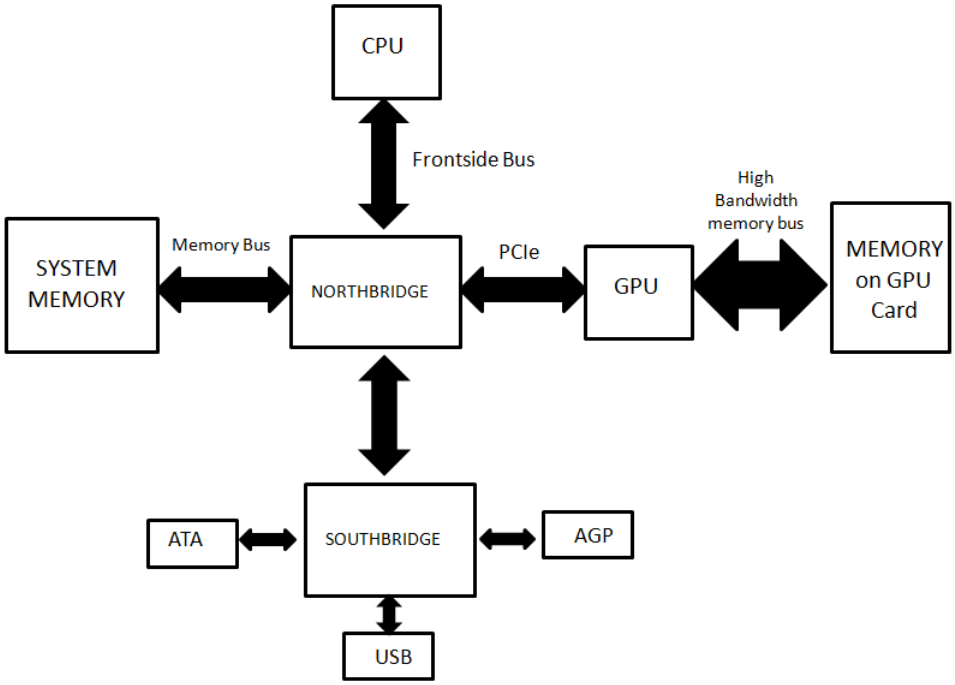

Fig. 17. System layout with the GPU connected via the PCIe bus

to its off-chip on-card memory through a high speed, high bandwidth memory bus. In order to enjoy performance lift from parallel execution in GPU, it is imperative to keep the data transfers across the PCIe bus to the GPU memory to a minimum. Fig.17 shows the schematic of a system configuration with a discrete graphics card connected at the PCIe slot. In our laboratory set up, an external 110W power module was connected to the motherboard to supply additional power to the graphics card.

1. CPU-GPU Communication and Control

It is critical to understand that in CPU-GPU interaction, the most expensive operation is the transfer of data from CPU-accessible memory to GPU-accessible-high-bandwidth local memory. To minimize this cost, it is important to design algorithms in a way that when data is transferred from CPU memory to GPU memory, enough operations are performed on the data to make the transfer worthwhile. If the memory transfer latency cannot be hidden, GPGPU execution may be more expensive than profitable. This is the chief constraint in using GPGPU for acceleration. Fig.18 shows how the GPU with many more ALU units than the CPU is capable of executing more data-parallel instructions per unit time, but the low bandwidth of the bus connecting the system memory and the graphics memory will define the actual performance gain. The peak bandwidth between the device memory and GPU on the 8800 GT card is 57.6 GBps while the peak bandwidth between host memory and device memory is 8 GBps on the PCIeX16. Therefore, for best performance of the application, it is important to minimize data transfer between the system and the graphics memory, even if that implies executing functions on the GPU that does not demonstrate any speed up compared to executing it on the CPU [34].

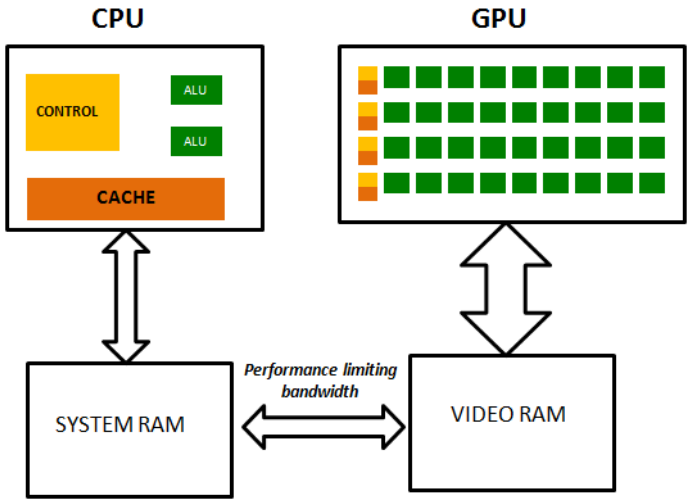

Fig. 18. Low bandwidth between system memory and graphics memory limits performance gain from parallel execution on GPU

2. Hardware Architecture

GeForce 8800 GT is the eighth generation of Nvidia's Tesla based graphics card equipped with a 65nm G92 general purpose graphics processing unit (GPU). It has 754 million transistors on chip.

Tables II and III list the hardware configuration of 8800 GT, obtained by executing a device query. 8800 GT has compute capability of 1.1. Table II lists the number of streaming cores and the maximum execution configuration of active threads, blocks and grids supported by 8800 GT. Table III specifies the core and memory clock frequencies and the classification of graphics memory into different sub-memory types. The bit-width of the GPU-graphics memory interface is 256, and the bandwidth of this bus is 57.6GBps.

Table II. Device attributes of 8800 GT

Major Version	1
Minor Version	1
Number of multiprocessors	14
Number of cores	112
Warp size	32
Maximum number of threads per block	512
Maximum sizes of each dimension of a block	512 x 512 x 64
Maximum sizes of each dimension of a grid	65535 x 65535 x 1
Maximum memory pitch	262144 bytes
Texture alignment	256 bytes
Concurrent copy and execution	No

Table III. Device attributes of GeForce 8800 GT card

Core clock	600MHz
Shader clock	1.5GHz
Memory clock	900MHz
RAM type	GDDR3
Total amount of global memory	512MB
Total amount of constant memory	64KB
Total amount of shared memory per block	16KB
Total number of registers available per block	8192
Memory interface	256bit
Memory bandwidth	57.6GB/s

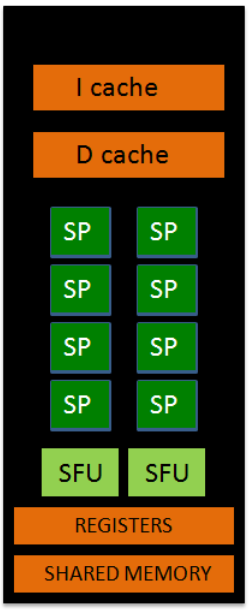

Fig. 19. A Streaming Multiprocessor in GeForce 8800 GT

Figure 19 depicts a streaming multiprocessor in the G80 Tesla architecture. The GPU device consists of 14 streaming multiprocessors (SMs), each containing 8 streaming processors (SPs), all running at 1.5GHz [35]. Each SM has 8192 32-bit registers that are shared among all threads assigned to the SM. The threads on a given SM's cores(SPs) execute in SIMT (single-instruction, multiple-thread) fashion, with the instruction unit broadcasting the current instruction to the eight SPs. Each SP has a single arithmetic unit that performs IEEE 754 single precision floating point arithmetic and 32-bit integer operations. Additionally, each SM has two special functional units (SFUs), which perform more complex floating point operations such as the trigonometric functions with low latency. The SPs and the SFUs are fully pipelined.

a. Peak Theoretical GFLOPS and Bandwidth

Each SM can perform 18 FLOP per clock cycle (2 FLOP per SP, 8SPs per SM and one complex operation per SFU, 2 SFUs per SM), yielding 378 GFLOPS (14 SM * 18 FLOP per clock cycle per SM * 1.5 GHz) of peak theoretical performance for the GPU. If we take into account the single-cycle multiply-add operations in the SPs alone, a more conservative GFLOPS measure is given by

$$Number of SPs * \frac{FLOP}{cycle * SP} * \frac{cycles}{s} = 112 * 2 * \frac{(1.5 * 10^9)}{(10^9)} GFLOPS = 336 GFLOPS \tag{3.1}$$

Nvidia reports a peak theoretical 504GFLOPS for GeForce 8800 GT [35]. Since operations taken into account for this definition are not known, we will use our own definition of GFLOPS given in Eq. 3.1 whenever we use it to evaluate performance.

8800 GT GPU has a 256-bit GDDR3 memory interface with a clock frequency of 900MHz. The theoretical bandwidth is given by

$$\frac{Bytes}{cycle} * \frac{cycles}{s} = \frac{(256 * 2)}{8} \frac{bytes}{cycle} * .9 \frac{Gcycles}{s} = 57.6 GBps \tag{3.2}$$

The multiplicative factor of 2 in Eq. 3.2 is because of double data rate of transfer of GDDR3 memory.

The bandwidth of 57.6 GB/s is the capacity of the memory interface connecting the GPU to to its 512 MB, off-chip, global memory. This off-chip memory has high transfer latency $\approx$ 8GBps compared to the on-chip memory on the GPU, hence memory transfers over this bus should be kept at minimum. Fig.19 shows the low-latency on-chip shared memory, instruction and data caches available on the device. Fig. 20 is a schematic of the GeForce 8800 GT card.

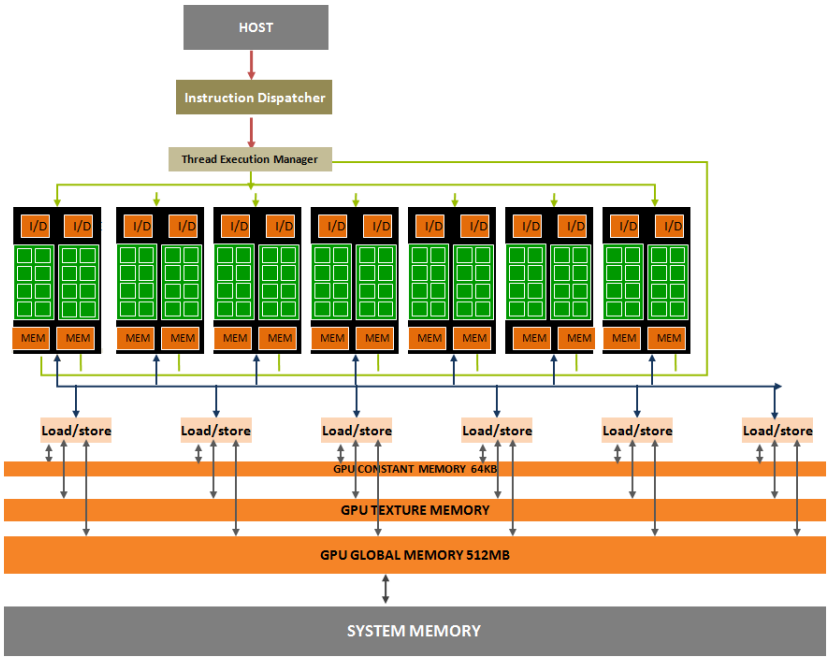

Fig. 20. Schematic of GeForce 8800 GT card

3. Programming Model

The G92 Tesla architecture supports the SIMT (single-instruction, multiple-thread) programming model. Tesla architecture supports NVIDIA's Compute Unified Device Architecture (CUDA) - a software and hardware interface for users to develop data parallel algorithms. The CUDA package consists of a driver, a run-time environment and fast utility libraries.

CUDA is a parallel execution model and threads are its units of execution. Threads in CUDA are organized into a three-level hierarchy. At the highest level, each kernel consists of many thread blocks. Each thread block consists of many threads, the maximum number of threads per block being 512. There is no upper limit on the number of blocks that can be launched on a grid. However, the configuration of blocks has to be judicious enough to warrant the cost of its launch. That is, if too many blocks with less than optimal threads are configured, during execution, each of these blocks will occupy an SM. The SM's resources such as registers, shared memory, SPs will have to be initialized, but if there aren't enough threads in a block, most of these resources will remain inactive throughout execution. The initialization costs are not justified in this case [30].

During execution, all threads in a block get launched on one SM i.e., threads from the same block cannot run on different SMs. Threads in the same block can share data through the shared memory and can perform barrier synchronization by invoking the inbuilt synchronization primitives. Thus, threads communicate and cooperate on the GPU. During execution, threads within a block are organized into warps which are groups of 32 threads. Warps are units of scheduling and threads are grouped into batches of warps for scheduling convenience only as it is not required by the underlying hardware. The configuration of threads in a block and blocks in a grid

need to be made by the developer at compile time and cannot change at run time. A thread has a local ID within a block and a global ID in the grid. Similarly, blocks have unique global IDs. These ids are used at run time for indexing data in the memory [30].

The CUDA programming interface consists of ANSI C extensions supported by several keywords and constructs. CUDA treats the GPU as a coprocessor that executes data-parallel kernel functions. The developer supplies a single source program that includes both host (CPU) and kernel (GPU) code. NVIDIA's compiler separates the host and kernel codes, which are then compiled by the host compiler and the kernel compiler respectively. The host code transfers data to and from the GPU's global memory via API calls, and initiates the kernel code by calling a function.

4. Memory Model

CUDA memory is organized into multiple hierarchies. There are registers and shared memory and data and instruction caches on-chip and local memory, texture, constant and global memory off-chip. The on-chip memories can exploit data locality and data sharing to reduce an application's demands for low bandwidth off-chip memory. From Table IV, we see that the GeForce 8800 GT has a 64 kB, off-chip constant memory, and each SM has an 8 kB constant memory cache. Because the cache is single-ported, simultaneous accesses of different addresses yield stalls. However, when multiple threads access the same address during the same cycle, the cache broadcasts that address's value to those threads. The latency of access from a cache broadcast is the same latency as that of register access. In addition to the constant memory cache, each SM has a 16 kB shared memory for data that is either written and reused or shared among threads. Finally, for read-only data that is shared by many threads but not necessarily accessed simultaneously by all threads, there are the off-chip texture

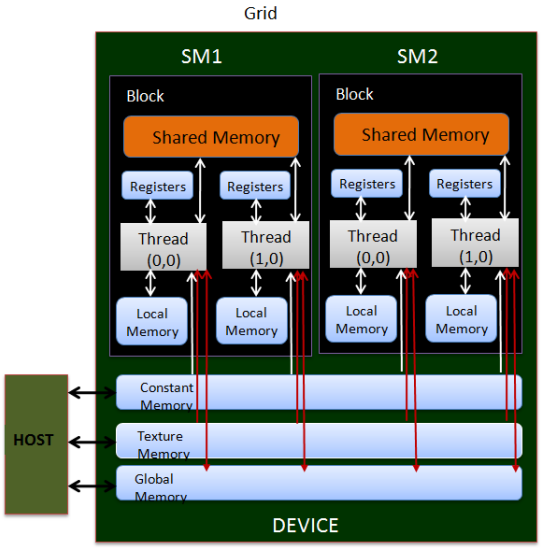

Fig. 21. Memory model in GPU computing

memory and the on-chip texture caches that exploit 2D data locality and reduce memory latency substantially.Global memory is uncached and cost of high latency access can be mitigated if data access is coalesced. Fig. 21 represents a schematic of the memory hierarchy in Tesla hardware and CUDA programming model.

Each thread has its own private local memory and its own registers. To share data within their containing block, threads use shared memory. However, local memory is accessed from off-chip global memory and hence involves high latency. Each SM has 8192 32-bit registers and a maximum of 768 threads can be launched on an SM with each block being able to contain a maximum of 512 threads. Blocks across SMs can read data from off-chip constant memory and texture memory, but do not share data. Blocks from all SMs can read from and write to global memory.

As can be seen in Fig. 21, the host can read and write into off-chip constant,

texture and global memory, but cannot access registers or shared memory on the GPU chip. Table IV lists the various memories available for programming and the latencies in each case.

5. Execution Model

Execution model of CUDA is thread based. The Thread Execution Manager efficiently schedules and coordinates the execution of thousands of computing threads. Each thread-warp executes in SIMD fashion, with the SM's instruction unit broadcasting the same instruction to the eight cores on four consecutive clock cycles. SMs can interleave warps on an instruction-by-instruction basis to hide the latency of stalled warps that are waiting on global memory accesses. When one warp stalls, the SM can quickly switch to a ready warp in the same thread block or in some other thread block assigned to the SM. The SM stalls only if there are no warps with all operands available. Threads in the same block can share data through the shared memory and can perform barrier synchronization by invoking the inbuilt synchronization primitives. Thus threads communicate and cooperate within their containing blocks and cannot communicate across blocks even if these blocks are in the same SM. Threads are otherwise independent, and synchronization across thread blocks is safely accomplished only by terminating the kernel [35].

C. Methodology

In this research, implementation of elastography algorithms is based on the use of cross-correlation algorithms for tissue motion estimation. Cross-correlation based estimators have narrow error margin, are sensitive and robust [9]. However, such accuracy comes with the price of heavy computational intensity and a high cost of

Table IV. Table of various memories on 8800 GT and their properties

Memory	Access	Scope	On/Off-Chip	Cached or Uncached	Amount	Latency
Registers	R/W	1Thread	On	n/a	8192/SM	1 cycle
Shared	R/W	1Block	On	n/a	16KB/SM	1 cycle
Local	R/W	1Thread	Off	Uncached	16KB/Th	≈1-300 cycle
Texture	R	Device & Host	Off	Cached	64KB	≈2-300 cycles
Texture cache	R	Device	On	n/a	8KB	1 cycle
Constant	R	Device & Host	Off	Cached	64KB	≈2-300 cycles
Constant cache	R	Device	On	n/a	8KB	1 cycle
Global memory	R/W	Device & Host	Off	Uncached	512MB	≈2-300 cycles

implementation.

Fortunately, the computational intensity is due to the iterative construct of the algorithm and the data processed in one iteration is independent of the data processed in another iteration. In the case of elastography, each A-line segment of the post compressed echo signal is cross-correlated with each A-line segment of the pre-compressed signal within a lateral search window, and this is repeated for all the A-lines and for the entire axial length of the A lines. It can be seen how the core operation is multiply and add of data points in the case of time-domain cross correlation and the product of fourier transforms of these data segments and its inverse fourier transform in the case of frequency domain cross correlation, and how it is the *repetition* of this operation over all the data points that accounts for maximum consumption of computational resources. It is in this characteristic that we see the opportunity for parallelism. *This data-independent iterative characteristic of the elastography algorithm maps impressively well to the Single Instruction Multiple Data model and may therefore make it possible to achieve significant speed-up when executed on the GPU.*

The axial displacement estimation function that uses cross-correlation for estimation is the first function offloaded to GPGPU for parallel execution. This takes care of the most computationally intensive part(≈51% of execution time) of the strain estimation algorithm. The remaining stages viz. temporal stretching(≈5% of execution time), median filtering (≈10% of execution time), interpolation(≈4.5% of execution time), searching for maximum peak (≈14.5% of execution time) and staggered strain estimation (0.1% of execution time) are also offloaded to GPU for execution in stages, and the performance benefit is measured. The distribution of execution time is with reference to Fig. 22. For more details regarding the stages of strain estimation elastography algorithm and the computational cost of each stage, please refer to the Methodology section in Chapter II. Memory optimizations are applied by implement-

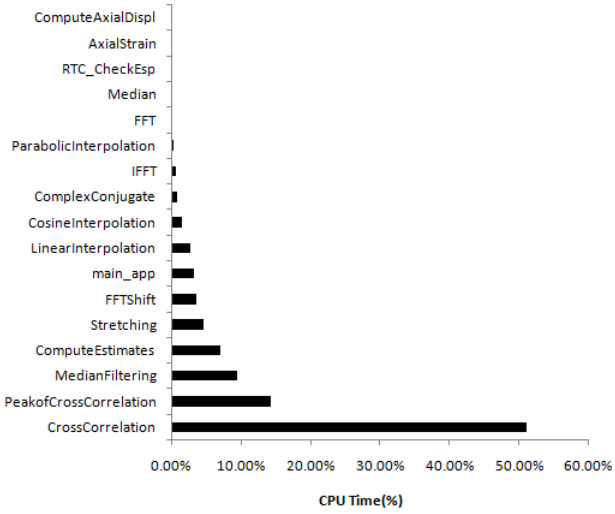

Fig. 22. Percentage of CPU time spent on different functions during the execution of elastography implemented in C

ing coalesced memory access, pinned memory, asynchronous copy, shared memory and texture caches. The gains at each stage are measured.

From Amdahl's law for parallel computing, the maximum speed up that can be expected from parallelizing sequential code is given by

$$S = \frac{1}{(1 - f_p) + \frac{f_p}{N}} \tag{3.3}$$

where

S : Maximum overall speed up of the application

f_p : Fraction of the code that was parallelized

N : Total number of cores executing in parallel

If N is very large, $\frac{f_p}{N} \approx 0$ and Eqn. 3.3 reduces to $S = \frac{1}{1-f_p}$. It can be seen that

S is now independent of N and will be significant if f_p is appreciably large. If f_p is small, increasing N will do little to influence any performance lift.

All runtime tests are performed on sample data obtained by simulating a medium (40 x 38 mm^2) with mechanical properties similar to those of soft tissues [36]. The simulation parameters are as follows:

- Center frequency of transducer : 6.6MHz
- Bandwidth : 50%
- Beamwidth : 1mm (at the focus)
- Scanning elements in the linear array transducer : 128
- Sampling frequency : 40MHz

These parameters match our diagnostic Ultrasound system. The sample is subjected to a constant axial compression under quasi-static conditions. The axial strain compression was varied in the range of 0.01%-10% in increments of 10.

The signal processing parameters are:

- Window length : 2.5% -7.5% of acquisition depth
- Window overlap : 80% Window length
- Stretching factor : equal to the applied strain

Based on the simulation parameters, the raw simulated RF frame has a dimension of 5195x128 samples. The resulting axial elastograms have dimensions in the range 95x128 - 65x128 depending on the selected window length.

Image quality tests were performed on two sets of simulated data. A simulated homogeneous phantom was used for the SNRe test, and a simulated medium containing one stiffer inclusion was used for the CNRe test. These are standard tests

for elastography In both the sets of experiments, additive Gaussian noise with SNR ranging from 0dB to 40dB was added to the RF data to simulate more realistic noisy conditions.

To measure the performance gain from execution on parallel GPU hardware, we offloaded the frequency domain cross-correlation based strain estimator for elastography to the GPU in 10 stages. The first 9 versions involve optimization toward memory bandwidth and GPU utilization and most of the computation is done on GPU while the CPU controls the flow. The tenth version is a hybrid version that configures both the CPU and GPU to compute at the same time, as peers. This hybrid version will allow efficient overlapping of waiting time with concurrent computation while processing continuous frames in real-time. The code transformation from C to CUDA for acceleration was done in collaboration with Xu Yang.

In the base version, we executed the major functional blocks in elastography (cross-correlation, median filtering, stretching and staggered strain) in data-parallel fashion on the GPU, without using even the simplest optimizations to conserve memory bandwidth or tolerate long latency loads or long latency trigonometric operations. In the second version, we optimized GPU occupancy by altering execution configuration of blocks and threads to maximize utilization of available computing resources. In the third version, we used pinned host memory to make more efficient use of the memory bandwidth. The fourth version uses coalesced global memory loads and stores to improve bandwidth. In the fifth version, we used shared memory to reduce memory latency. In the sixth version, we improved performance by reducing divergent branches. The seventh version reduces avoidable data transfers. In the eighth version, we implemented parallel reduction to register performance lift. In the ninth version, we employed loop unrolling for better speeds. In the tenth version,we implemented a hybrid computation scheme where both CPU and GPU performed computation

concurrently, hiding waiting time in any of the devices and making efficient use of available resources.

The GPU implementation was done using CUDA version 2.0 interfaces and libraries. The kernel code was compiled using CUDA compiler nvcc with ptx flag and O3 and -fast_math optimization switches. CUDAProfiler v2.0 was used to analyse performance and identify hotspots such as uncoalesced memory access, warp serialization or low GPU occupancy.

The speed performance data (execution time, GFLOPS) was obtained by processing the simulated data five times with each of the 7 implementations of the cross correlation based elastography algorithm and the average performance was reported. For the image quality test, we executed 3 runs each of the GPU.Coalesce version and the CPU-only version and compared the average values.

D. Results

1. Speed Up

Our implementation of parallel elastography using GPU is 67x faster than its corresponding sequential version that uses CPU for computation. The latest version of parallel elastography takes .052s to generate an axial elastogram (19.23fps) for a pair of given pre- and post compressed data and signal processing parameters specified in the Methodology section. For the same input, the C implementation takes 3.5s (.285fps). The speed up is bound by the CUFFT library that is executed for $\approx$40% of the time. The performance we have reported here is conservative as we test with input data and signal processing parameters that are higher than ones used in clinical diagnosis. For typical diagnostic depths and elastographic processing parameters, this implementation can be predicted to yield elastographic frame rates in the order

of 50fps. The following sections describe how we accelerate the algorithm through incremental optimizations. GFLOPS, GBps, GPU Time and CPU time were used to evaluate computational performance.

a. GPU.Base

The GPU.Base is the first step in parallelizing elastography. 4 stages of the elastography algorithm -temporal stretching, axial displacement estimation, median filtering and strain estimation are offloaded to the GPU for execution. To implement a parallel cross-correlation estimator for computing axial displacement, we use the CUFFT library available in the CUDA developer package. CUFFT is built on FFTW module developed by Matteo Frigo and Steven G. Johnson at MIT [37].

No optimization is attempted at this stage. This version of the implementation has frequent data transfer between host memory and graphics memory. From Table V, we see that the average bandwidth between system memory and graphics memory is ≈1.4GBps. Frequent memory transfers over this low bandwidth bus is a major performance bottleneck. The net speed up in execution time achieved in this parallel version with respect to the sequential C version is 2.5x.

b. GPU.Occupancy

Global memory access latencies are approximately 300 cycles long. One way to hide global memory access latency is to overlap it with active execution in the GPU. To keep the GPU utilization high, if a particular warp in an SM is waiting for global memory access, other ready warps can execute to hide the memory access latency. Therefore, we need to have high SM occupancy. *Occupancy* refers to the ratio of the number of active warps to the total number of warps per multiprocessor [30]. The occupancy of a multiprocessor should be close to 1 to ensure that there are enough active

warps and hence the multiprocessor is never idle. At the same time, occupancy higher than 1 will cause resource conflicts and performance will be less than optimal. In this version, we maximize memory latency hiding by choosing an execution configuration that maximizes multiprocessor occupancy. Several configurations of blocks per grid and threads per block are tried and the performance is recorded. By increasing GPU occupancy of previously low-occupancy kernels by 4 times, we achieve a performance lift of 1.52x. Further increasing the occupancy does not improve performance.

c. GPU.PinnedMemory

Pinned system memory is memory that is locked against paging. When a portion of physical memory is pinned, it is not used for paging. Therefore, data residing on pinned memory is assured to be always available as long as the application associated with the data is scheduled. Memory transfer between host and device is improved by using pinned memory and the bandwidth of such transfers can attain over 5GBps [34]. Table V and Table VI show the maximum host-device and device-host bandwidth achievable with pageable memory and pinned memory on our test configuration. Fig. 23 compares the bandwidth available with paged memory and pinned memory. CUDA memory copy is synchronous but usage of pinned memory makes asynchronous transfers possible. The DMA engine on the graphics card controls memory transfers between graphics memory and pinned system memory and transfers data asynchronously while CPU does useful work. For data to be allocated on pinned memory on 8800 GT, it has to be 64 byte aligned [30]. Pinned memory usage improves performance of elastography by 1.465x.

Though constant and texture memories are cached, the read-only access restriction limits their usage in our application.

Table V. Paged memory bandwidth on GeForce 8800GT

Device to Device Bandwidth	46.906 GB/s
Host to Device Bandwidth	1.588 GB/s
Device to Host Bandwidth	1.228 GB/s

Table VI. Pinned memory bandwidth on GeForce 8800GT

Device to Device Bandwidth	45.988 GB/s
Host to Device Bandwidth	2.422 GB/s
Device to Host Bandwidth	1.547 GB/s

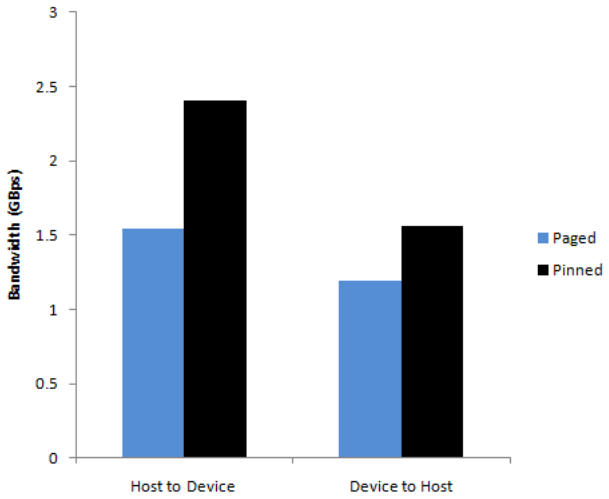

Fig. 23. Comparison of bandwidth obtained using paged memory versus pinned memory on 8800GT and E4500 Core2 Duo

d. GPU.Coalesced

Global memory bandwidth is used most efficiently when simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be coalesced into a single memory transaction. Global memory accesses are coalesced when threads in half a warp access data sequentially and the accessed data is a multiple of 4, 8 or 16 bytes [30]. If the accesses are non-sequential or memory unaligned, bandwidth is 8 times lower than coalesced accesses for single floating-point data. This is because when access is uncoalesced, 32 bytes of data, the minimum transaction size will be fetched of which only 4 bytes of data is used by a thread in the half-warp resulting in $32/4 = 8$x degradation in performance. By allocating graphics memory using proper CUDA functions, the access to RF data array is assured to be coalesced since memory is allocated with a pitch that is 16 times the size of the accessed data (for 64byte alignment specifications) and its rows are padded accordingly. With this optimization, GPU.Coalesced runs 2.77x faster than GPU.PinnedMemory. Non-unit strided access costs 15 additional transactions in half a warp, hence caution is taken to not effect any non-unit strided access.

e. GPU.SharedMemory

Shared memory accesses have single cycle latency compared to ≈300 cycle latency global memory accesses. Shared memory can be used as a software cache for data in the global memory that cannot be uncoalesced. Hence we can expect to achieve performance benefit by copying data from global memory to shared memory if we can also avoid shared memory bank conflicts. Bank conflicts occur when threads in half a warp access data from the same bank. In this case, the memory accesses are serialized and the effective bandwidth degrades. Also, shared memory is limited to

16kB per block, hence not all required data can be brought from global memory to shared memory. An optimal choice needs to be made on the amount of repeatedly used data that should be copied from global memory to shared memory. By adopting bank-conflict free shared memory accesses in our implementation, the performance of GPU.SharedMemory improved 1.2x with respect to GPU.Coalesced.

f. GPU.ReducedDivergence

Flow control instructions cause threads in the same warp to diverge and hurt the instruction throughput. When threads in a warp diverge, the non-divergent threads have to wait for the divergent threads to traverse their execution paths and then converge. Effectively, execution paths are serialized, increasing the total number of instructions executed for this warp while non-divergent threads waste time waiting for the divergent threads to converge. In flow controls where the test condition is based on the thread id, the impact of divergence was reduced by changing the control condition. A speed up of 1.375x was achieved over GPU.SharedMemory.

g. GPU.ReducedDataTrf

Pinned memory is a limited resource and cannot be used for all data transfers. Hence to reduce the cost of data transfers, we make the best use of available pageable memory by grouping data transfers. The cost of one large data transfer is lesser than the cost of many small data transfers [34]. By changing the code to reduce data transfer time, we exceed the performance gain from GPU.ReducedDivergence by 1.33x.

h. GPU.ParallelReduction

Parallel reduction is a programming technique that allows computation of partial results by parallel threads which are ultimately combined to the final result in $\log_2(n)$

steps instead of (n-1)steps. Reduction is well suited for linear algebra operations that add and/or multiply n values or compute the minimum or maximum of n values. Using parallel reduction to accelerate sub-functions inside axial displacement estimation, GPU.ParallelReduction registers a speedup of 1.06x over GPU.ReducedDataTrf.

i. GPU.LoopUnroll

In parallel reduction, branching overheads can be reduced by unrolling loops. When the number of effective threads is less than or equal to the size of a warp, there is no need to test each thread for the branching condition since we know that all the remaining threads will take the branch. This allows us to unroll the last six loops in vector reduction method, and achieve more instruction throughput and memory bandwidth. In GPU.Unroll, loop unrolling reaches a speed up of 1.9x with respect to GPU.ParallelReduction. The advantage of loop unrolling can be exploited by unrolling loop for the whole kernel using CUDA templates [29].

j. Hybrid Computation

In this version we modify the computing scheme from a master-slave model to a collaborative model where the CPU is responsible not only for control and coordination, but also shares computing load with the GPU.

In the implementation of elastography, median filtering does not benefit from GPU execution as much as axial displacement estimation does. This is primarily due to the way the median filter algorithm accesses data. Each iteration of the filtering algorithm requires data within the radius of the filter kernel in two dimensions. This leads to huge number of uncoalesced loads from the global memory. Shared memory usage causes bank conflicts and does not provide improvement over uncoalesced global memory loads. Also we cannot use the advantage of 2D locality of the texture cache

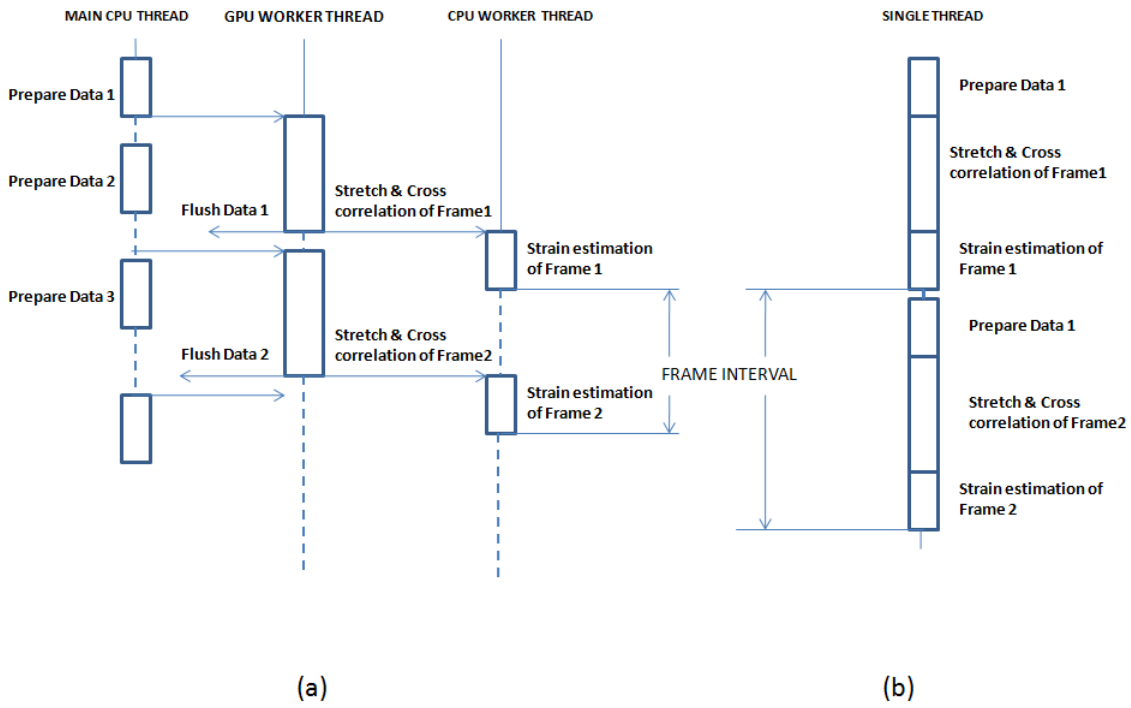

Fig. 24. Sequence flow in elastography (a) When both CPU and GPU are used for computation (b)When only GPU is used for computation

Table VII. Summary of speed ups in axial strain elastography using different optimizations

Version	Execution time	Speedup
CPU	3.5	-
GPU.Base	1.4	2.5x
GPU.Occupancy	.92	1.52x
GPU.PinnedMemory	.638	1.44x
GPU.CoalescedAccess	.23	2.77x
GPU.SharedMemory	.192	1.2x
GPU.ReducedDivergence	.14	1.375x
GPU.ReducedDataTrf	.105	1.33x
GPU.ParallelReduction	.099	1.06x
GPU.LoopUnroll	0.052	1.9x
GPU.Hybrid	0.036	1.42x

or constant cache since data has to be first copied from global memory to texture or constant memory and this costs GPU cycles. Additionally, the median search within the kernel causes costly branching and divergent warps. Since median filtering precedes strain estimation, if we decide to perform median filtering on the CPU, the strain estimation should also be executed on the CPU to minimize CPU-GPU data transfer.

Nevertheless, there is still data transfer that has to happen from graphics memory to system memory. Processed axial displacement estimates need to be transferred to

the system memory for median filtering and strain estimation stages to execute. This transfer time can be used to our advantage if we already have data for the next frame in the GPU while the CPU performs filtering and strain estimation on the previous frame. The transfer time can be efficiently overlapped with concurrent computation on the CPU and GPU. Thus, while the nth frame is being transferred from graphics memory to system memory, GPU computes axial displacement for the (n+1)th frame while CPU computes the strain for the (n-1)th frame. Such a design flow maximizes computational resource utilization and will manifest as improved throughput. This scheme is optimally suited for real-time processing where data is acquired as a stream of frames and processed images have to be generated as a stream of frames. Hybrid computation gives a speed up of 1.33 times when multiple frames are processed. Since this approach requires multiple frames to be in flight, the hybrid approach might not show any improvement over an implementation that needs to process only individual frames at a time. Fig.24 compares the sequence flow of the elastography algorithm in a GPU-compute and CPU-GPU hybrid compute model.

Table VII and Fig.25 show the incremental speedups in execution time achieved by employing the optimization techniques discussed above.

Table VIII and Fig.26 show the speedups in throughput achieved in the four main stages of processing elastograms offloaded to GPU for execution.

2. Image Quality

In this section, we discuss the comparison of image quality of elastograms generated with GPU execution with those generated with CPU execution. Tables IX, X, XI

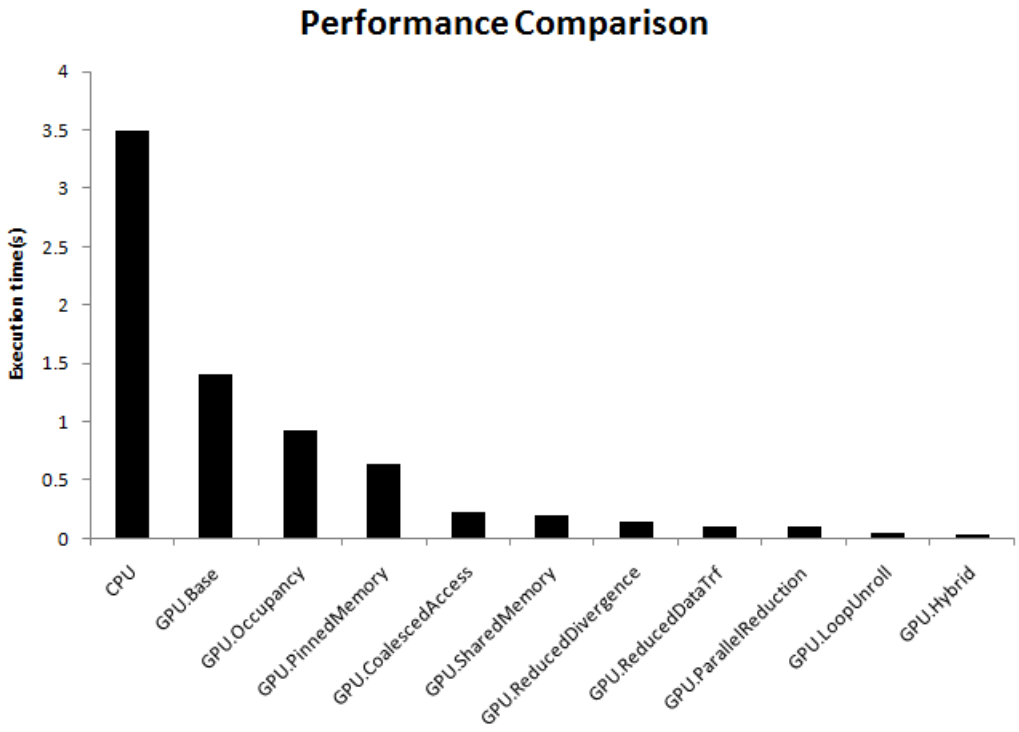

Fig. 25. Time cost for generating axial strain elastograms using different versions of code

and XII summarize the results of the image quality study. In each table, we compare the statistical quality of the test image generated by the CPU only version with the CPU-GPU version. In Tables IX and XII, we compare the CNRe of non-homogeneous targets with different background noise and different window lengths respectively. In Tables X and XII, we compare the SNRe of homogeneous targets with different background noise and different window lengths respectively. Figures 27 and 28 show the strain filters associated with the GPU and CPU implementations.

The above image quality analysis statistically proves that GPU accelerated elastograms have the same quality as CPU generated elastograms. at there is no sta-

Table VIII. Table of speed ups in GFLOPS different stages of elastography

Function	C GFLOPS	CUDA GFLOPS	Speedup
Stretching	.016419	1.285732	78.3x
AxialDisplacement	.029091	3.224145	110.83x
MedianFiltering	.013359	.347048	25.978x
AxialStrain	0.02431	1.295492	53.29x

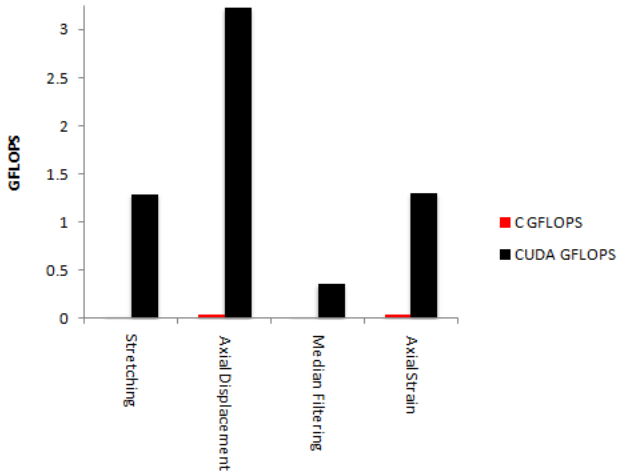

Fig. 26. GFLOPS speed up achieved from executing functions in GPGPU

Table IX. Table of CNRe of CPU and GPU computed elastograms when different background noise is applied

	CNR_e at 40dB SNR_s		CNR_e at 20dB SNR_s		CNR_e at 10dB SNR_s	
Strain%	CPU	GPU	CPU	GPU	CPU	GPU
.01	1.44	1.44	1.0403	1.0403	1.0398	1.0398
.1	2.04	2.04	2.0292	2.0292	1.0195	1.0195
1	29.6461	29.6461	29.2917	29.2917	27.2791	27.2791
2	65.793	65.793	64.5928	64.5928	63.0665	63.0665
10	1.0143	1.0143	1.0523	1.0523	1.0516	1.0516

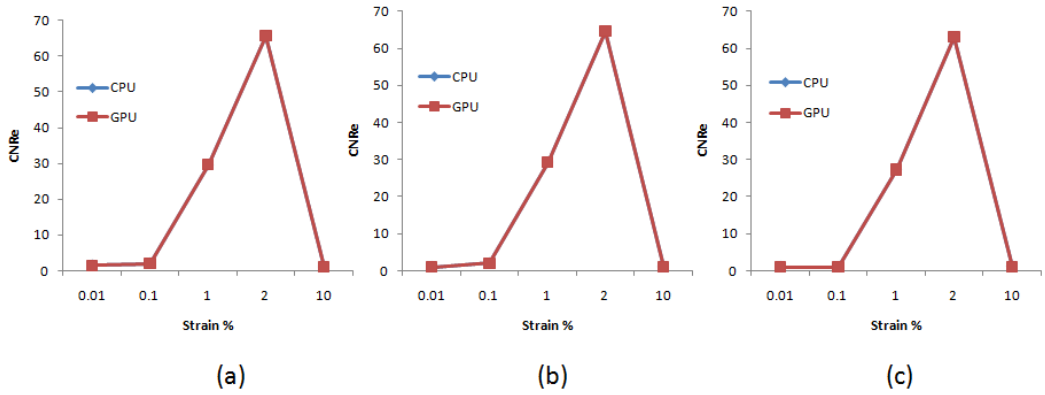

Fig. 27. Graph of CNRe on CPU and GPU generated elastograms at (a)40dB SNRs (b) 20dB SNRs (c) 10dB SNRs

Table X. Table of SNRe of CPU and GPU computed elastograms when different background noise is applied

	SNR_e at 40dB SNR_s		SNR_e at 20dB SNR_s		SNR_e at 10dB SNR_s	
Strain%	CPU	GPU	CPU	GPU	CPU	GPU
.01	1.565	1.621	1.583	1.583	1.673	1.683
.1	1.743	1.749	2.003	2.005	1.648	1.647
1	8.2144	8.2144	8.1322	8.1322	7.5363	7.5482
2	104.4075	104.4074	113.7135	113.7135	109.8109	109.8109
10	1.3277	1.3272	1.3181	1.3289	1.2978	1.3037

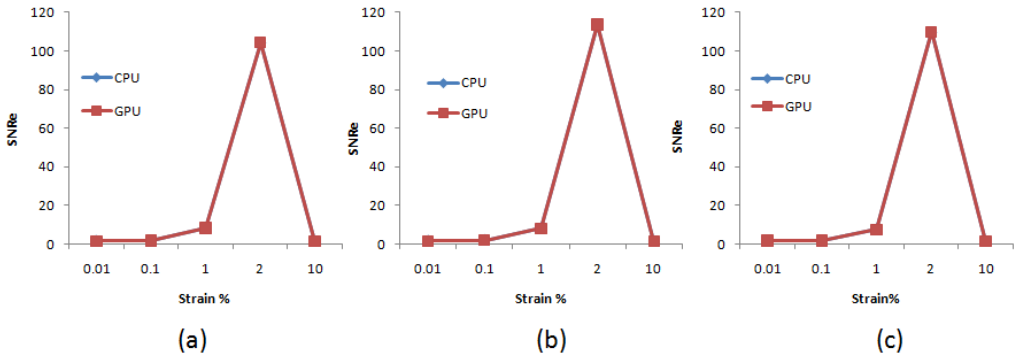

Fig. 28. Graph of SNRe of CPU and GPU generated elastograms at (a)40dB SNRs (b)20dB SNRs (c)10dB SNRs

Table XI. Table of CNRe of CPU and GPU computed elastograms at different window lengths

	CNR_e at 40dB SNR_s					
	w = 2.5%		w = 5%		w = 7.5%	
Strain%	CPU	GPU	CPU	GPU	CPU	GPU
.01	1.22	1.22	1.044	1.044	1.0112	1.0112
.1	2.602	2.602	2.04	2.04	2.0264	2.0264
1	8.2638	8.2638	29.6461	29.6461	34.6078	34.6078
2	45.6851	45.6851	65.7493	65.7493	76.0812	76.0812
10	1.0174	1.0174	1.0143	1.0143	1.1424	1.1424

Table XII. Table of SNRe of CPU and GPU computed elastograms at different window lengths

	SNR_e at 40dB SNR_s					
	w = 2.5%		w = 5%		w = 7.5%	
Strain%	CPU	GPU	CPU	GPU	CPU	GPU
.01	1.3109	1.3109	1.1565	1.1621	1.9251	1.9251
.1	2.3109	2.3109	2.1743	2.1749	2.8483	2.8483
1	4.923	4.923	8.2144	8.2144	9.7108	9.7108
2	41.2328	41.2328	104.4075	104.4075	204.5518	204.5519
10	1.2733	1.2733	1.3277	1.3272	1.3314	1.3314

tistically significant difference in the analyzed image quality factors between CPU elastograms and GPGPU elastograms. For the purpose of illustration, Fig.29 shows a set of elastograms generated by CPU and GPU.

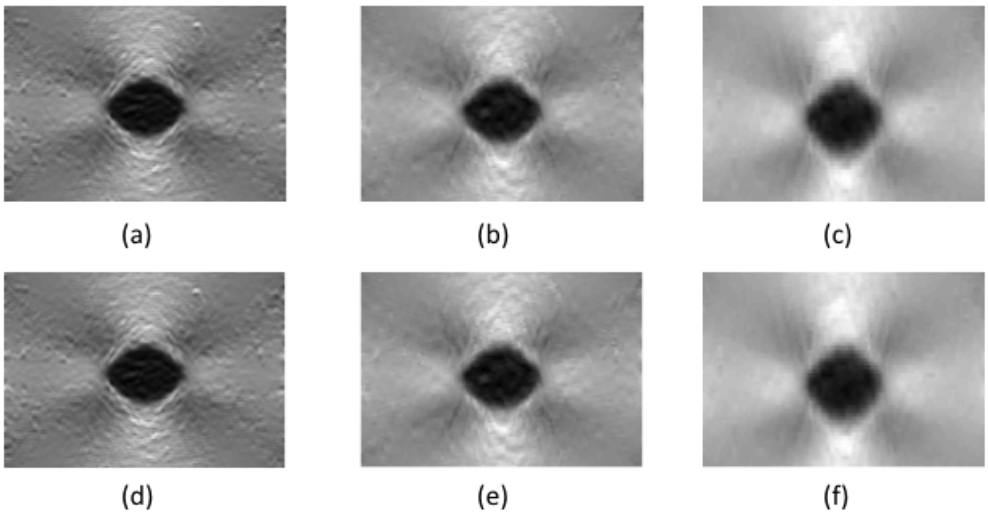

Fig. 29. (a-c)GPU generated elastograms generated at 1mm, 2mm and 3mm window length. (d-f)CPU generated elastograms generated by the CPU algorithm at same values of window length

a. Experimental Validation

In addition to simulations, the performance of GPU generated vs CPU generated elastograms was also tested on a set of experimental RF data obtained from a tissue mimicking phantom. Fig. 30 shows the elastogram of the phantom generated by the GPU and the CPU.

These experimental results corroborate the simulation findings that image quality is uncompromised by GPU acceleration.

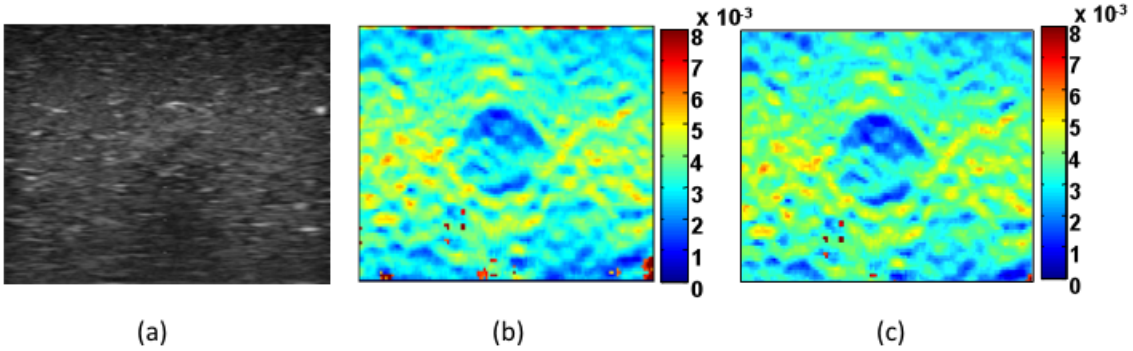

Fig. 30. Images of a tissue mimicking phantom containing a stiff inclusion.(a)B mode sonogram (b)Elastogram on CPU (c)Elastogram on GPU

E. Performance Analysis

Detailed profiling using CUDAProfiler 2.0 reveals bottlenecks such as uncoalesced loads and stores, divergent branching, warp serialization and host-device transfer bandwidth that impose a bound on the performance of the current implementation. Figures 31 through 36 provide detailed information about the performance of parallel elastography on GPU.

Profile of individual functions and their performance blocks are shown in figures 38 through 43.

F. Discussion

By exploiting parallelism in the GPU and using available computational resources between the CPU and GPU efficiently, we accelerated sequential elastography 67 times, without loss in image quality, to yield a frame rate of ≈19.23fps. If we use the CPU-GPU hybrid model, we can reach a frame rate of 27.77 fps. Since our

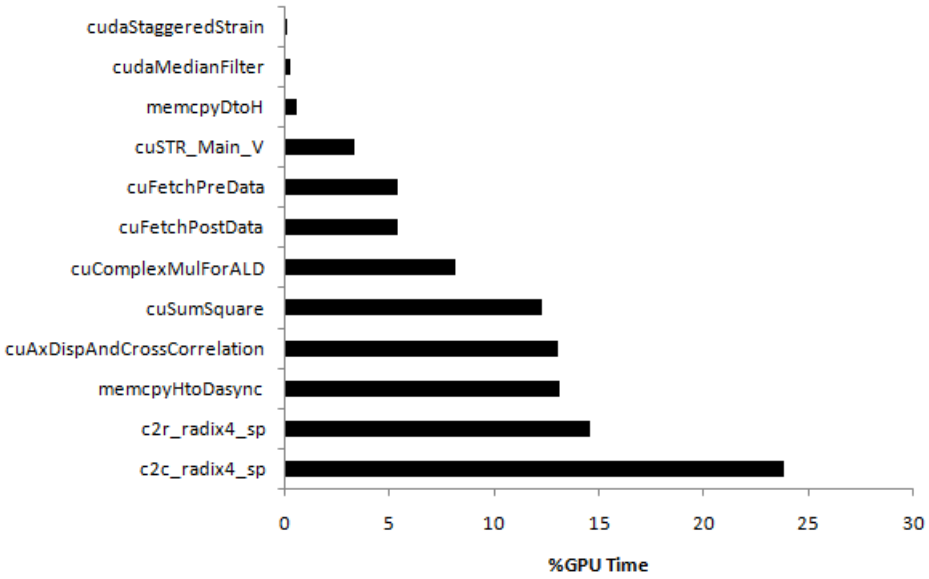

Fig. 31. GPU time profile of strain estimation algorithm

measurements are conservative in the context of input data and signal processing parameters typically used in clinical diagnosis, we can expect this implementation to deliver a peak frame rate in the order of 50fps.

In terms of GPU utilization, with a 73.95 GFLOPS and given G92's theoretical peak performance at 336 GFLOPS (Refer section Peak GFLOPS in Chapter II), we can expect that this implementation can be further improved. From the %GPU profile in Fig.31, we can see that most of the GPU time is consumed by c2c_radix4_sp and c2r_radix4_sp which are fft routines in CUFFT libary. The performance of the current implementation of parallel elastography is limited by the performance of the CUFFT library. From figures 32,33,34,35 and 36, we see that CUFFT has high uncoalesced global memory loads, high divergent branching, high warp serialization and low occupancy, each of which is a performance degrading factor. NVIDIA reports a

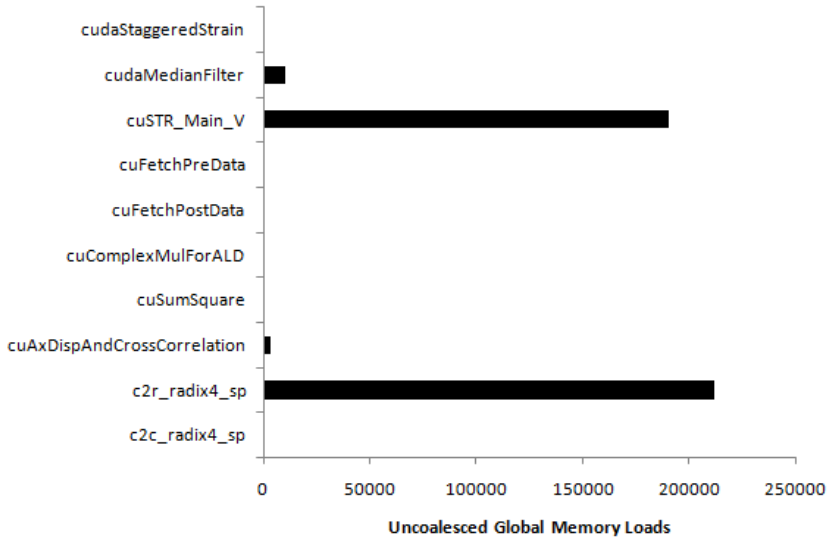

Fig. 32. Profile of uncoalesced loads from global memory during strain estimation on GPU

throughput of 52GFLOPS for CUFFT on 8800GTX [37], while our axial displacement estimation function that uses CUFFT achieves a throughput of 3.22GFLOPS. The difference in performance is possibly due to 2 reasons. First, 8800GTX with 128 parallel cores and 86.4GBps is a higher performance card compared to 8800GT with 112 parallel cores and 57.6GBps [29]. Second, the difference between actual performance and peak performance may be due to difference in the number of FFT points chosen for computation.

We could not use the low-latency on-chip texture and constant caches to our advantage since they are read-only memories and our implementation requires frequent writes. Consequently, all intermediate outputs were written to the global memory and this added significantly to the program latency.

From Fig.32 and 33, we note that our implementation of temporal stretching

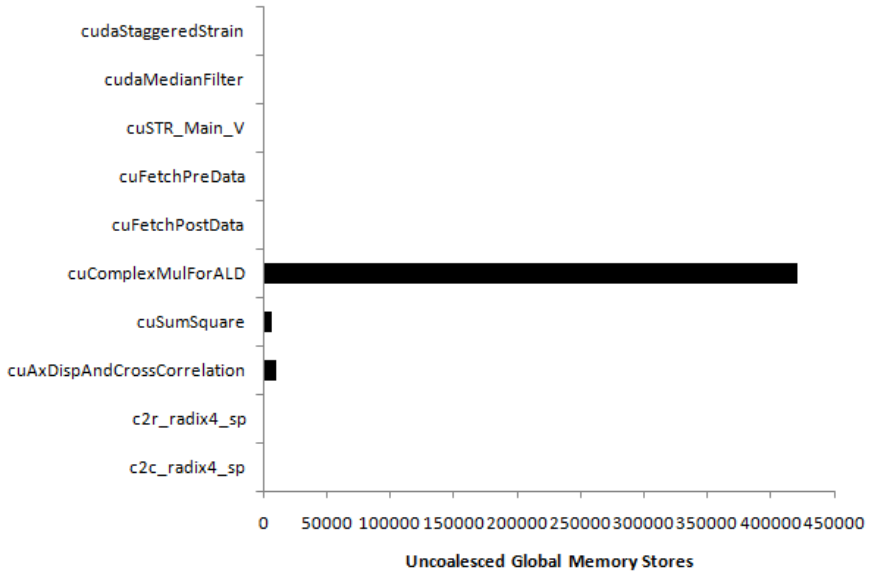

Fig. 33. Profile of uncoalesced stores to global memory during strain estimation on GPU

and complex multiplication suffer from high uncoalesced loads and stores respectively. This is because threads in the stretching function process data along a column and memory loads cannot be coalesced. This problem can be remedied if data layout is changed in a way that threads access contiguous data along a row. Transposing the data in the graphics memory with respect to the system memory may lead to coalesced memory loads. Similarly, in the case of complex multiplication, data storage has to be improved to reduce random access of global memory locations.

High divergent branching in cuSumSquare and cuAxDispAndCrossCorrelation in Fig.38 and 39 suggest that flow control conditions need to be modified better in these functions to reduce branching overheads. In both these functions, we also notice relatively high warp serialization in Fig.35 which suggests that shared memory bank

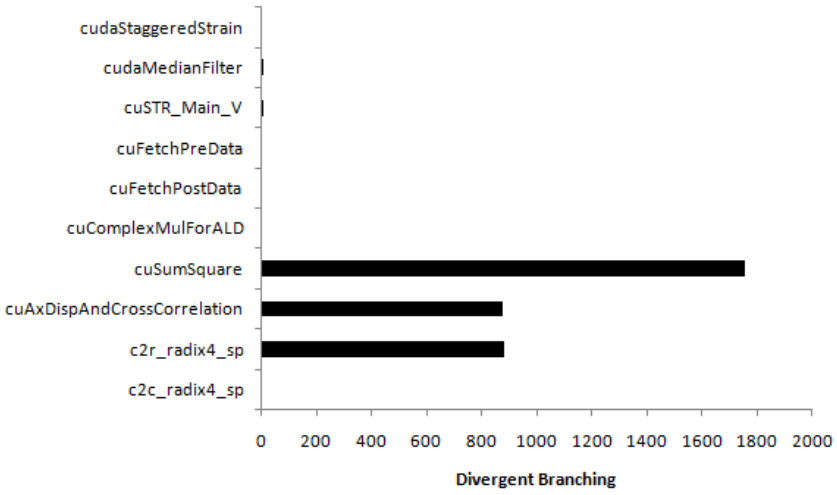

Fig. 34. Profile of divergent branching in strain estimation on GPU

conflicts have to be resolved better in these functions.

From Fig.36, we note that temporal stretching, complex multiplication, median filtering and staggered strain estimation have relatively low occupancy. To make better utilization of computing resources in the GPU and justify initialization costs, it is important to improve the occupancy in these functions. Fig.37 shows the ratio of time spent on CPU execution to the time spent on GPU execution for each function.

About 14% of GPU time is still spent in device to host and host to device memory transfers. It needs to be investigated if this can be overlapped with execution either via asynchronous copy or staged copy and execution [34].

Finally, not all stages in elastography map well to SIMD model for GPGPU programming. The relatively poor GFLOPS speed up in median filtering shows how sub-optimal mapping may be inevitable in certain algorithms. The median filtering algorithm would perform better if it could be fetched from the texture cache that is

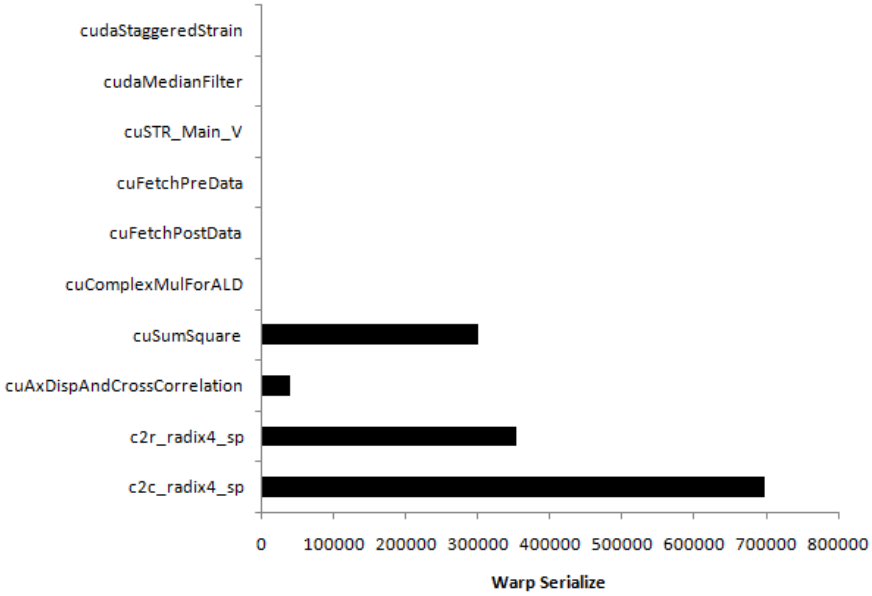

Fig. 35. Profile of warp serialization in GPU strain estimation

optimized for 2D locality. However, texture is read-only and median filtering being an intermediate stage, it would cost more to write to the first texture from the host and then fetch from the cache. We could consider using the shared memory for faster access, but shared memory is limited and the implementation in shared memory cannot be generalized for all input data sizes. Thus, inspite of having computational resources, we are not able to use them to our benefit due to the inherent structure of the algorithm. There is, however, always a possibility of engineering a scheme to overcome such challenges through thorough investigation.

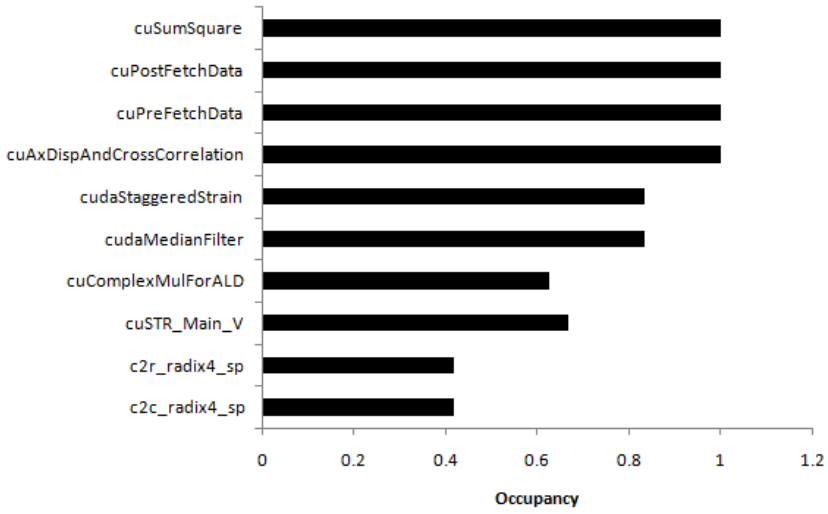

Fig. 36. Occupancy profile in strain estimation on GPU

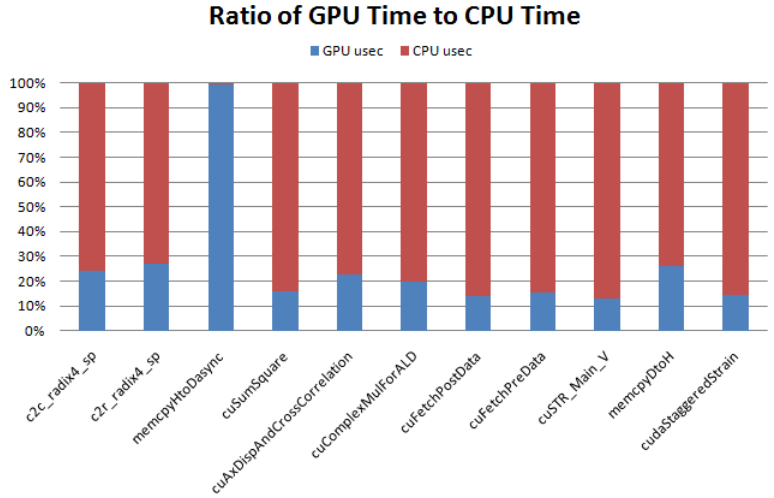

Fig. 37. Ratio of CPU to GPU time for various functions

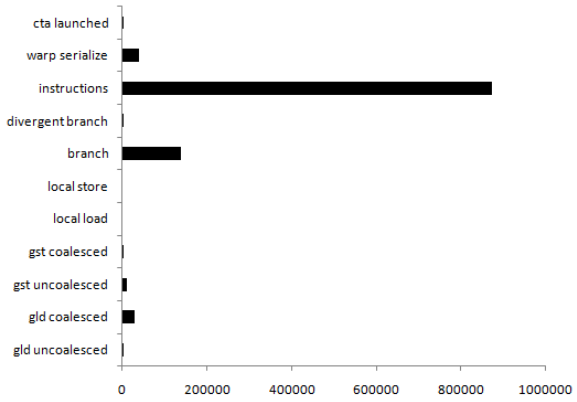

Fig. 38. Profile of axial displacement estimation on GPU

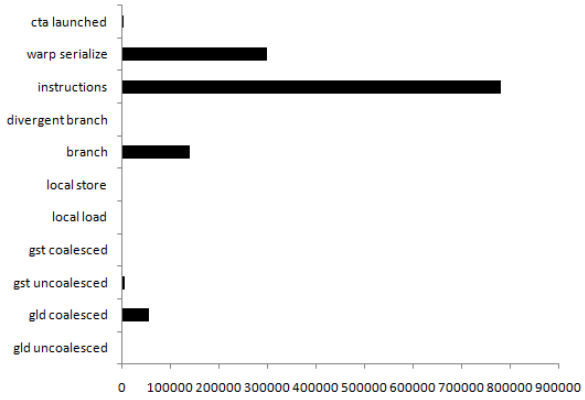

Fig. 39. Profile of sum of squares computation on GPU

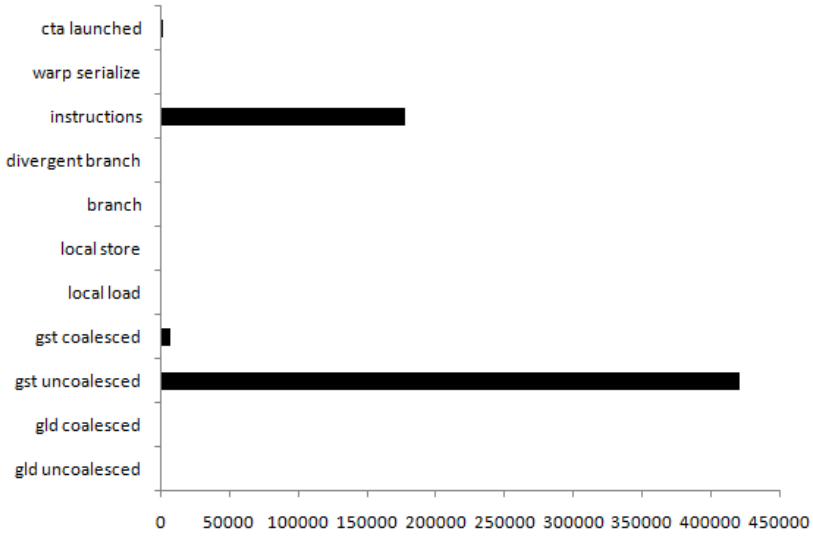

Fig. 40. Profile of Complex Multiplication on GPU

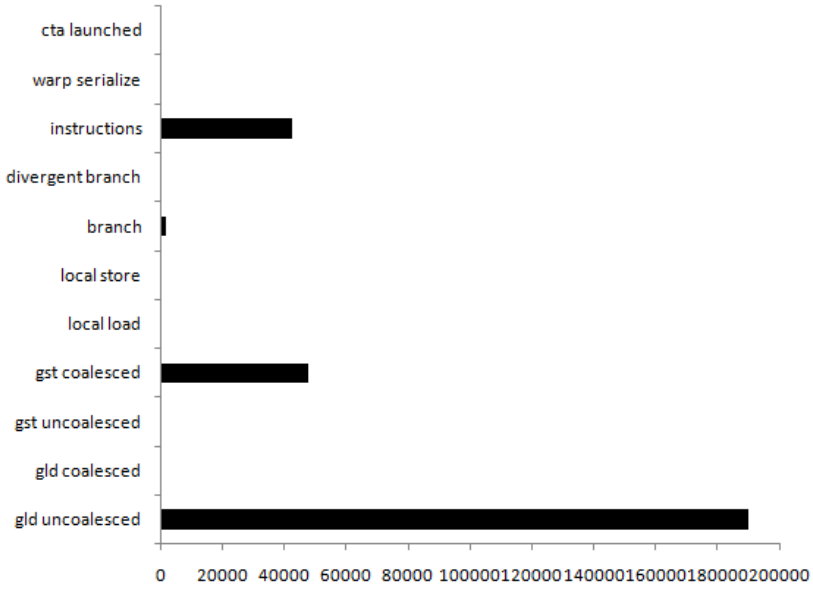

Fig. 41. Profile of temporal stretching on GPU

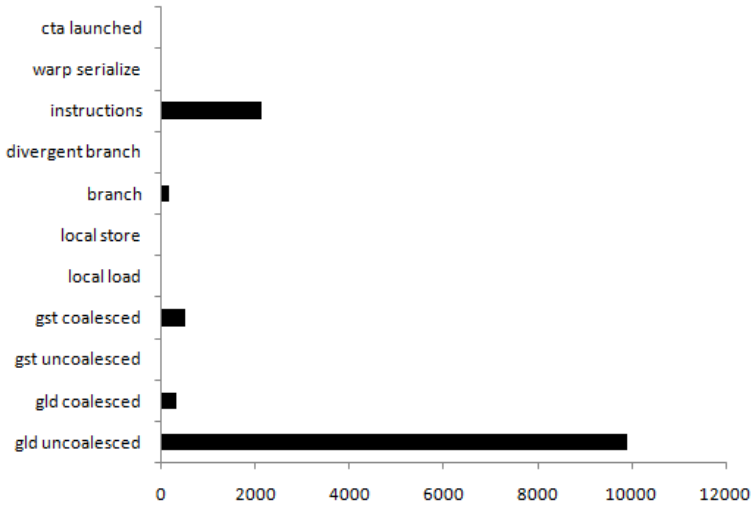

Fig. 42. Profile of median filtering on GPU

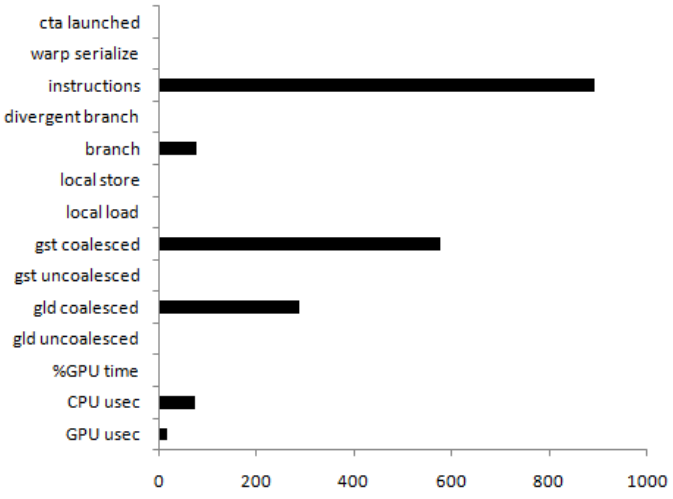

Fig. 43. Profile of staggered strain estimation on GPU

CHAPTER IV

SUMMARY AND FUTURE WORK

A. Summary of Research

Using GPU as a co-computational device to the CPU, we have demonstrated that cross correlation based elastography can be accelerated to deliver real time performance with no statistical loss in image quality. There are limitations to parallelizing cross-correlation based elastography because of certain inherent data-dependencies and unavoidable random memory acceses in the algorithm. The implementation should scale transparently to better hardware. A multi-threaded version of our software executing on a 4GPU G200 based Quadro FX 5600, can be expected to generate elastograms at 4 times the frame rate achievable with a single GPU G92 based GeForce 8800 GT.

The heterogeneous approach to computing where CPU, GPU and other compute devices are alloted computational load that conform well to their underlying architecture appears to allow achieving performance optimization at little extra cost. The OpenCL standard [38] and the availability of its libraries can make load-balancing through heterogeneous computing a reality.

B. Avenues for Future Work

Since GPU based elastography does not have any precedence, realization of the concept has been incremental and has gone through rounds of iterations and there still remains a wide scope for improvement in the implementation. There are several ways in which the memory and computational efficiency of our solution can be enhanced. We elaborate on a few of these opportunities for improvement.

1. Improving CPU and GPU Utilization

In the current implementation, CPU usage is pinned at 50% and is using only one of the two cores. The implementation can be modified to have its computational load equally distributed among the active cores in the CPU. With such modification, we can expect the performance to scale and have an even better CPU-GPU heterogeneous solution. As seen in Fig. 36, there is scope for improving GPU utilization by improving the occupancy of relevant functions.

2. Using Faster FFTW Libraries

We used fftw 2.x for this implementation. The latest fftw 3.x delivers higher GFLOPS. However, it is backward incompatible with 2.x, hence we did not change the implementation. This can be taken up as future work. Also, at the time this report was written, the FFT implementation by Govindaraju et.al [39] reported performance $\approx$4 times faster than CUFFT. This implementation could not be used due to lack of time, and should be taken up as future work.

3. Making the Algorithm Compute Bound

An implementation whose performance is not defined by memory transfers but by computational latency is said to be compute bound. A compute bound implementation has fully utilized the resources available, and further improvement is possible only on improved hardware. Two standard metrics used for evaluating performance are

- Memory Warp Parallelism (MWP): Measure of how many warps' memory requests can be bundled together at one instant of time
- Compute Warp Parallelism (CWP): Measure of how many warps are in execu-

tion at the same time. CWP is similar to compute intensity

If MWP $\geq$ CWP, the implementation is memory bound and there is scope for improvement. By making memory load-stores coalesced, by avoiding shared memory bank conflicts, the performance can be enhanced. If MWP $\leq$ CWP, the implementation is at its best and is compute bound. Another way to analyse performance is by comparing the number of active warps to MWP. If number of active warps in the implementation is less than the MWP, the implementation should be changed to increase the number of warps since increased number of threads improve processor utilization. If however, the number of active warps has exceeded the MWP, the processor is fully utilized and there is no further scope for improvement.

In short, for a compute bound algorithm the following relation should be used as a guideline.

$$MWP \leq Warps \leq CWP \tag{4.1}$$

When this inequality is ensured, performance can be predicted.

In our implementation, memory transfers account for roughly 15% of total GPU time. If this can be hidden by overlapping computations, we can have a truly compute bound implementation that can be further accelerated only with better hardware.

4. CPU-GPU Load Balancing

For optimal performance, a CPU-GPU collaborative programming paradigm is most promising. An automatic load balancing scheme where the runtime decides if a code segment can be sent to the GPU for computation, or if the GPU is running at 100% utilization, the CPU should handle the computation, is elegant because it is adaptive to the computational environment and stands to gain from knowing the current utilization and capacity of the active devices. A static allocation would suffice if the

input load is not dynamic and the performance of the participating devices is known from previous instrumentation. However, for dynamic loads as in real-time applications, a dynamic load balancing scheme will be indispensible. To implement dynamic load balancing, a performance estimator will be required. The performance estimator will provide initial inputs to the load balancing scheme. Performance models have been proposed in the academia to estimate and predict GPU performance. A performance estimator should take into account thread occupancy, memory access trends, amount of memory warp parallelism, amount of compute warp parallelism, number of registers used, amount of shared memory used, warp serialization etc. Using these inputs, the performance estimator should be able to give an initial estimate of the execution time and the cycles per instruction per warp.

With these initial inputs, the load balancer will dispense the first computational load to the active devices. Once initialized, the load balancer will wait for feedbacks from performance counters monitoring performance of each of these computing devices. These counters can provide information about resource utilization, cache performance, power consumption

A CPU-GPU load balanced solution will provide optimal performance.

5. Integrated CPU-GPU

Precious CPU cycles are wasted in copying data from CPU to GPU and back when DMA transfer is not possible due to physical memory constraints. Also in devices that do not support concurrent copy and execution, these operations are forced to stall. An integrated CPU-GPU platform that can obviate the need for extra copy will improve performance as the time gained from avoiding copy can be utilized in doing useful work.

6. Speedup Using Multiple GPUs

To increase processing power, Nvidia endorses the Scalable Link Interface (SLI) standard which allows multi-GPUs to share workload in parallel processing and produce a single output. ATI offers CrossFire as a competing solution. It will be interesting to see how elastography scales with multiple-GPUs and what trade-offs are inherent in a multi-gpu solution.

C. Conclusion

Our preliminary work proves the feasibility of accelerating cross-correlation based strain estimator by mapping it to the GPGPU computing platform. With accelerated performance, we are able to benefit from the sensitivity of cross-correlation estimator which is crucial in real-time elastography. We hope to extend the current implementation to accelerate a wide range of estimators that require fast and efficient solutions to elastography.

We would like to explore the possibility of executing these algorithms on heterogeneous computing platforms equipped with load-balancing feature to gain superior speed-quality performance.

Real time 3D elastography is a promising area of medical imaging that can benefit immensely from parallel computing for accelerated performance. Our results prove that the returns on investing in parallelizing 3D elastography may be very high. It will also be exciting to be able to compute multiple elastograms like axial strain elastogram, lateral strain elastogram, moduli elastogram, time-constant elastogram, Poisson's ratio elastogram and display them on the same screen. With more information from multiple elastograms, the clinician will be able to make better decisions. This will contribute toward precise, accurate and objective diagnosis while also re-

ducing patient-cycle time.

REFERENCES

[1] J. Ophir, I. Cespedes, B. Garra, H. Ponnekanti, Y. Huang, and N. Maklad, "Elastography - Ultrasonic imaging of tissue strain and elastic modulus in vivo," *European Journal of Ultrasound*, vol. 3, pp. 49–70, 1996.

[2] J. Ophir, K. Alam K, B. Garra, F. Kallel, E. E. Konofagou, T. A. Krouskop, and T. Varghese, "Elastography: Ultrasonic estimation and imaging of the elastic properties of tissues," *Proceedings of the Institution of Mechanical Engineers*, pp. 1–31, 1999.

[3] E. E. Konofagou, T. P. Harrigan, J. Ophir, and T. A. Krouskop, "Poroelastography: Imaging the poroelastic properties of tissues," *Ultrasound in Med. & Biol.*, vol. 27, pp. 1387–1397, 2001.

[4] Unmin Bae and Yongmin Kim, "Real-time ultrasound elastography," *Proc. SPIE*, vol. 6513, 2007.

[5] Giorgio Rizzatto, "Real-time elastography of the breast in clinical practice: the Italian experience," MEDIX Suppl., 2007.

[6] http://en.wikipedia.org/wiki/Elastography, "Elastography," accessed in January 2010.

[7] J. Ophir, I. Cespedes, H. Ponnekanti, Y. Yazdi, and X. Li, "Elastography: A method for imaging elasticity of biological tissues," *Ultrasonic Imaging*, vol. 13, pp. 111–134, 1991.

[8] www.gpgpu.org/developer, "GPGPU developer resources," accessed in January 2010.

[9] E. I. Cespedes, "Elastography: Imaging of biological tissue elasticity," Ph.D. dissertation, University of Houston, Houston, TX, 1993.

[10] T. Varghese and J. Ophir, "Performance optimization in Elastography: Multi-compression with temporal stretching," *Ultrasonic Imaging.*, vol. 18, pp. 193–214, 1996.

[11] P. Chaturvedi, M. Insana, and T. Hall, "Analysis of breast lesions using elastography: Initial clinical results," *Radiology*, vol. 202, pp. 79–86, 1997.

[12] E. E. Konofagou, J. Ophir, T. A. Krouskop, and B. S. Garra, "Elastography : From theory to clinical applications," Key Biscayne, FL, June 2003, Summer Bioengineering Conference.

[13] A. M. Lubinski, S.Y. Emelianov, K. R. Raghavan, A. E. Yagle, A. R. Skovoroda, and M. O'Donnell, "Lateral displacement estimation using tissue compressibility," *IEEE Trans. Ultrason. Ferroelec. Freq. Control*, vol. 43, pp. 247–256, 1996.

[14] P. Chaturvedi, M. Insana, and T. Hall, "2D companding for noise reduction in strain imaging," *IEEE Trans. Ultrason. Ferroelect. Freq. Control*, vol. 45, pp. 179–191, 1998.

[15] P. Chaturvedi, M. Insana, and T. Hall, "Testing the limitations of 2D companding for noise reduction in strain imaging using phantoms," *IEEE Trans. Ultrason. Ferroelect. Freq. Control*, vol. 45, pp. 1022–1031, 1998.

[16] E. E. Konofagou and J. Ophir, "A new elastographic method for estimation and imaging of lateral displacements, lateral strains, corrected axial strains and Poisson's ratios in tissues," *Ultrasound in Med. & Biol.*, vol. 24, pp. 1183–1199, 1998.

[17] R. Righetti, J. Ophir, and P. Ktonas, "Axial resolution in elastography," *Ultrasound in Med. & Biol.*, vol. 28, pp. 101–113, 2002.

[18] T. Varghese, J. Ophir, E. E. Konofagou, F. Kallel, and R. Righetti, "Tradeoffs between the axial resolution and the signal-to-noise ratio in Elastography," *Ultrasound in Med. & Biol.*, vol. 29, pp. 847–866, 2001.

[19] T. Varghese and J. Ophir, "Estimating tissue strain from signal decorrelation using the cross-correlation coefficient," *Ultrasound in Med. & Biol.*, vol. 22, pp. 1249–1254, 1996.

[20] E. E. Konofagou, T. Varghese, and J. Ophir, "Spectral estimators in elastography," *Ultrasonics*, vol. 38, pp. 412–416, 2000.

[21] M. O'Donnell, A. R. Skovoroda, and B. M. Shapo, "Measurement of arterial wall motion using fourier based speckle tracking algorithms," in *Proceedings of Ultrasonics Symposium*, Orlando, FL, December 1991, IEEE Press.

[22] J. S. Bendat and A. G. Piersol, *Random Data: Analysis and Measurement Procedures*, John Wiley and Sons, New York, 3rd edition, 2000.

[23] S. K. Alam and J. Ophir, "Reduction of signal decorrelation from mechanical compression of tissues by temporal stretching: Applications to elastography," *Ultrasound Med. & Biol.*, vol. 23, pp. 95–105, August 1997.

[24] R. K. Palenichka, "Parallel median filtering algorithms and their real time implementation," Plenum Publishing Corporation, September 1989.

[25] Ben Weiss, "Fast median and bilateral filtering," in *ACM SIGGRAPH*, Boston, MA, 2006, ACM.

[26] S. Srinivasan, J. Ophir, and S. K. Alam, "Elastographic imaging using staggered strain estimates," *Ultrasound in Med. & Biol.*, vol. 24, pp. 224–245, 2002.

[27] T. Varghese and J. Ophir, "Enhancement of echo signal correlation in elastography using temporal stretching," *IEEE Trans. Ultrason. Ferroelect. Freq. Control*, vol. 44, pp. 173–180, 1997.

[28] F. Kallel, C. D. Prihoda, and J. Ophir, "Contrast-transfer efficiency for continuously varying tissue moduli: Simulation and phantom validation," *Ultrasound in Med. & Biol.*, vol. 8, pp. 1115–1125, 2001.

[29] www.developer.nvidia.com, "GPU computing source," accessed in January 2010.

[30] Nvidia, *CUDA Compute Unified Device Architecture Programming Guide v2.0*, Nvidia Corporation, Santa Clara, CA, June 2008.

[31] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics*, vol. 26, pp. 80–113, 2007.

[32] J. E. Lindop, G. M. Treece, A. H. Gee, and R. W. Prager, "3D elastography using freehand ultrasound with a 3D probe," *Ultrasound in Med. & Biol.*, vol. 32, pp. 463–474, 2006.

[33] S. Srinivasan and J. Ophir, "A zero-crossing strain estimator in elastography," *Ultrasound in Med. & Biol.*, vol. 29, pp. 227–238, 2002.

[34] Nvidia, *CUDA C Programming Best Practices Guide, CUDA Toolkit 2.3*, Nvidia Corporation, Santa Clara, CA, July 2009.

[35] Nvidia, *NVIDIA GeForce 8800 GPU Architecture Overview*, Nvidia Corporation, Santa Clara, CA, November 2006.

[36] T. A. Krouskop, T. M. Wheeler, F. Kallel, B. S. Garra, and T. Hall, "Elastic Moduli of breast and prostate tissues under compression," *Ultrasonic Imaging*, vol. 20, pp. 260–274, 1998.

[37] Nvidia, *CUDA CUFFT Library v1.1*, Nvidia Corporation, Santa Clara, CA, October 2007.

[38] www.khronos.org/opencl, "OpenCL overview," accessed in January 2010.

[39] N. K. Govindaraju, B. Llyod, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors," in *Proceedings of the IEEE Conference on Supercomputing*, Austin, TX, 2008, IEEE Press.

[40] T. Varghese and J. Ophir, "An analysis of elastographic contrast-to-noise ratio performance," *Ultrasound in Med. & Biol.*, vol. 24, pp. 915–924, 1998.

[41] J. D. Gaskill, *Linear Systems, Fourier Systems and Optics*, John Wiley and Sons, New York, 1978.

[42] S. K. Alam, J. Ophir, and T. Varghese, "Elastographic axial resolution: An experimental study," *IEEE Trans. Ultrason. Ferroelect. Freq. Control*, vol. 47, pp. 304–309, January 2000.

APPENDIX A

IMAGE QUALITY MEASUREMENT

The following standard metrics are used in elastographic image quality analysis.

1. SNRe
2. CNRe
3. Resolution
4. Sensitivity
5. Dynamic Range

An overview of these metrics is provided in this appendix.

1. SNRe

SNRe (elastographic Signal to Noise ratio) measures the accuracy and precision of strain estimate quantitatively as

$$SNRe = \frac{\mu_s}{\sigma_s} \tag{A.1}$$

SNRe is a measure of the performance of the strain estimator. Here

μ_s is the mean strain estimate

σ_s is the standard deviation of the strain estimate

Strain filter is the upper bound on SNRe obtained by replacing μ in A.1 by ideal tissue strain μ_t and by replacing σ_s by the theoretical lower bound on standard deviation $\sigma_L B$ and is represented as

$$SNRe^{UB} = \frac{\mu_t}{\sigma_{LB}} \tag{A.2}$$

$SNRe^{UB}$ is the upper bound on the performance of the strain estimator. This is the Strain Filter. It filters out the strain values/range in which SNRe drops lower than the threshold limit.

2. CNRe

CNRe (elastographic Contrast-to-Noise ratio) is a measure of the ability of the system to discriminate between two regions with different stiffness constant. It reflects on the ability of the imaging system to detect lesions. Mathematically, it is the ratio of the strain contrast between a region of interest and its background, to the noise in the system. Statistically, CNRe of inclusions with simple geometries (1D or 2D inclusions in uniformly elastic background) can be estimated by combining the properties of elastic modulus contrast (CTE), the US system physical parameters and the signal processing parameters defined by the Strain filter. An theoretical upper bound on the CNRe [40] is given by

$$CNRe^{UB} = \frac{2(\mu_1 - \mu_2)^2}{{\sigma_1}^2 + {\sigma_2}^2} \tag{A.3}$$

Here

μ_1: mean strain in region 1

μ_2: mean strain in region 2

σ_1: standard deviation of the strain in region 1

σ_2: standard deviation of the strain in region 2

From eq.A.3, we see that CNRe improves when the strain difference between the two regions increases and the total noise decreases. Upper bound CNRe improves at low modulus contrasts [2]. This is because the variances are low though the strain contrasts may not be high, yielding increased CNRe. At high modulus contrasts, the strain contrasts are high and hence the CNRe improves. In the middle region, CNRe drops because the mean strain differences are low. Finally, if the variances are high, even if the modulus contrasts are high, we get low CNRes. From the plot, for a given US system and signal processing parameters, CNRe can be maximized by tuning the strain. Maximum CNRe can be reached at relatively low strains of .5 to 1 percent when motion resulting from compression is least complex. Deviation of experimental values of CNRe from their theoretical estimates at high strains can be attributed to decorrelation in the elevational plane that has not been taken into account in this theoretical model.

3. RESOLUTION

One of the key parameters used to measure image quality and characterize the performance of an imaging system is image resolution [17]. Knowledge of the limits and trade-offs of resolution is fundamental to evaluating the feasibility of an imaging modality. In layman terms, resolution is the property of an image that describes the level of detail that can be discerned from it. In technical terms, resolution is defined as a measure of the ability of the system to discern between two closely spaced point sources [41]. The exact definition of resolution is application specific. In sonography,

resolution is defined the smallest distance that can be resolved between two reflecting boundaries. If the resolution is measured in the direction of incident waves, the resultant axial resolution is given by -

$$AR = \frac{Q\lambda}{4} \tag{A.4}$$

Here

AR: Axial Resolution

Q : Quality factor of the transducer = 1/fractional Bandwidth at a given λ

λ: Wavelength of the incident acoustic waves = $\frac{c}{f_c}$

c : Velocity of wave in the medium

f_c: Frequency of the transducer

With Q factor being constant for a given imaging system, the above equation shows that AR can be increased by decreasing λ or increasing the center frequency of the transducer. If Q factor can be controlled, then AR can be improved by increasing the fractional Bandwidth of the transducer.

Elastography uses the same definition for resolution as is used in sonography The earliest study performed on elastographic axial resolution concluded that it is equivalent to the cross-correlation window length [9], [40]. Later studies by [42] showed that elastographic axial resolution is a function of both the window shift Δw and the window length w, with Δw being more important than w. By means of a simulation experiment, it was shown that resolution varies linearly as the window shift Δw, for a given w and for values of Δw lower than a threshold, the resolution is insensitive to DSP parameter tuning. In [17], it was demonstrated through a 2D simulation experiment that elastographic axial resolution is limited fundamentally by the physical wave parameters of the US system used to acquire data, though

non-optimal choice of DSP parameters will compromise the best attainable axial resolution. Axial resolution was shown to be of the order of the ultrasonic wavelength, as in sonography.

4. SENSITIVITY

In elastography, sensitivity, S_e is the smallest strain that can be detected at a given SNRe. Sensitivity is the minimum strain estimation error that can be achieved in an elastography system. Strain estimation error is related to time delay estimation by

$$\sigma_s = \sqrt{2}\frac{\sigma_t}{\Delta w} \tag{A.5}$$

Here,

σ_s = error in strain estimation

σ_t = error in time delay estimaton

Δ_w = window shift

According to definition,

$$S_e = \sigma_{smin} = \sqrt{2}\frac{\sigma_{tmin}}{\Delta w} \tag{A.6}$$

From Eq. A.6, we see that the sensitivity depends on the window shift Δw. As we increase the window shift, the sensitivity improves. But with increased window shift, the spatial resolution deteriorates. This is the trade-off between sensitivity and resolution.

5. DYNAMIC RANGE

Dynamic range of an elastography system is defined as the ratio of its maximum measurable strain to its sensitivity. The dynamic range in decibels is given as

$$DR = 20 \log \frac{strain_{max}}{S_e} \tag{A.7}$$

Maximum strain in a homogeneous medium measurable by a system is defined as that strain at which 50% of the strain estimates are within 50% of the applied strain [9]. The maximum measurable strain has strong dependence on the quality of the signal processing algorithm used. In a heterogeneous target, applying a low strain would result in an elastogram that correctly detects the strains in the soft areas, but is insensitive to the low strains in the stiffer areas. Application of high strain would corrupt the image in the soft areas. Therefore, a compromise is reached on the maximum applied strain such that the soft areas suffer strain higher than the sensitivity and hard areas suffer strains lower than the sensitivity. The dynamic range of elastography is estimated to be on the order of 60dB. Systems have to be calibrated to accomodate the human visible range of approximately 40dB of dynamic range in gray scale images [9].

VITA

Sthiti Deka graduated with a B.Tech in Electrical and Electronics Engineering from the National Institute of Technology, Tiruchirappalli in India in 2004. She then joined Philips Medical Systems at Bangalore, India where she worked as a software engineer for three years. There she gained expertise in storage and retrieval of medical images and was exposed to challenges in multi-modality imaging. In 2007, she joined Texas A&M University to pursue an M.S. in Electrical and Computer Engineering and joined the Ultrasound Imaging laboratory at TAMU to work on tissue elastography. From January 2009 to December 2009, she worked with Advanced Micro Devices as a Co-Op Engineer in the Media Engineering Group. There she researched optimization opportunities on CPU-GPU integrated platforms for AMD's future processors. She enjoys tuning software applications for improved performance and is enthusiastic about CPU-GPU heterogeneous computing. She completed her M.S. in May 2010.

Address:
Department of Electrical and Computer Engineering,
Texas A&M University,
214 Zachry Engineering Center,
TAMU 3128,
College Station, Texas 77843-3128