Accelerating Ultrasound Elasticity Imaging with CUDA-MATLAB Approach

Rashid Al Mukaddim, Michael Turney, Robert Pohlman
Departments of Medical Physics & Electrical and Computer Engineering

ME759 Final Project Report
University of Wisconsin-Madison
December 21, 2017

Abstract

Ultrasound Elastography is a well-established field of study in medicine providing applications in many health areas. Elastography involves displacement estimation between ultrasound data, which can be achieved using methods such as two dimensional multilevel strain estimation [1]. The Multilevel Algorithm has many opportunities where parallel computing can boost performance. This project aimed to create a GPU framework for the Multilevel Algorithm and further optimize this framework to improve the performance of parallel execution. The final GPU Multilevel implementation is consistent with the CPU implementation with speedups of 8x – 20x.

Contents

1. Motivation

Elastography is a process of using Ultrasound to measure the elasticity of a material from an applied force. This has many important uses, especially in medical imaging. In our Ultrasound Lab in the Medical Physics Department, we work on calculating the elasticity for cardiac and liver applications. Figure 1 shows a representative case for the liver application. This done by calculating displacement from a given force in the tissue. The gradient of the displacement is then taken to give a relative strain measurement of the tissue. This strain is an important aspect for clinicians to deduce the tissue properties for assessing current treatments or ailments.

Current methods we use for calculating the displacement include the Multilevel using 2D Normalized Cross Correlation [1]; the displacement estimation method developed in our lab. The Multilevel algorithm operates as a pyramid, where initial displacements are computed starting on a coarse grid using large kernels, followed by computations on finer and finer spatial grids to obtain high SNR estimates with high spatial resolution. This method is very computationally expensive and time consuming, so it currently is done post-procedure due the time constraints of procedures. Similar processes to the Multilevel method have been implemented [2-5].

We aim to create our own GPU implementation of this displacement estimation method in hopes of speeding up processing time, leading towards a real-time implementation.

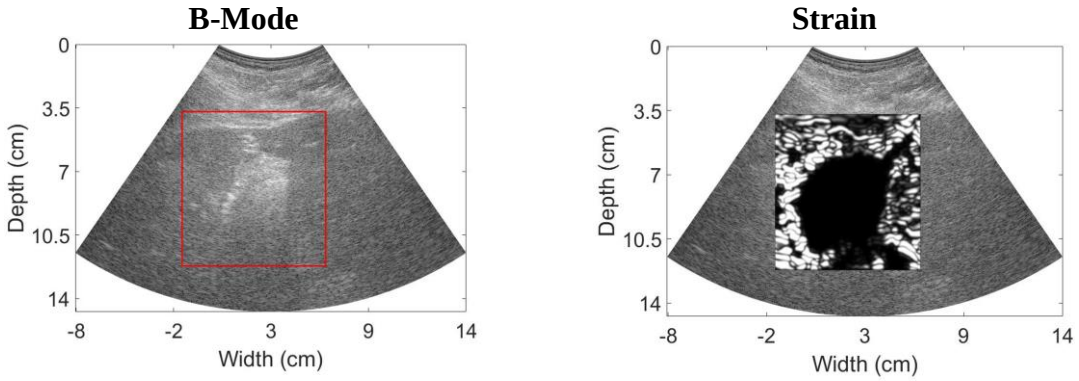


Figure 1 – Sector Ultrasound image of a liver after Microwave ablation procedure. Left image shows B-mode image with box surrounding ablated region, right image demonstrates the strain image of the boxed region.

2. Problem Statement

We aim to create a GPU implementation for one grid level of the Multilevel using 2D Normalized Cross Correlation to speed up displacement estimation computational time. This single level can easily be repeated in the future for the other levels to create the pyramidal scheme that the Multilevel ensues.

3. Implementation

3.1 Hardware Architecture

For this project, we chose to use the hardware available in our research lab instead of the Euler cluster from SBEL. Our lab has an NVIDIA TESLA K40 GPU with compute capability 3.5. One of the most notable areas of improvement with the Kepler architecture was power efficiency. The optimization and hardware changes can deliver up to 3x more performance per Watt compared to Fermi. Some additional differences between the Fermi and Kepler are outlined in Table 1.

Table 1 - [6]

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

3.2 CUDA-MATLAB Approach for Elastography

3.2.1 Principles of Elastography

An elastic medium under compression of uniaxial stress experience a cumulative strain principally along the axis of compression [7]. If some points in the medium has varying stiffness parameter than the others, then strain experienced among different points will be different resulting in a contrast through strain [7]. In Ultrasound Elastography, these strain contrast is derived through the analysis of ultrasound signals obtained from standard medical scanner. To accomplish this, ultrasound radio-frequency signals from region-of-interest is acquired without any applied compression (pre-compression frame) and then again with a small amount of compression along axis of insonification (post-compression frame). Pre and post compression frames are subdivided into 2-D windows and displacement is calculated through comparison of windowed signal using normalized cross-correlation technique. Finally, strain is estimated through least-squares fit of estimated inter-frame displacement [8]. The key stages involved in the strain estimation algorithm are – 1) Appropriate padding of pre and post-compression frames, 2) NCC computation, 3) Integer displacement estimation though peak finding of cross-correlation results, 4) Sub-sample displacement estimation using quadratic surface fit and 5) Strain estimation using least-squares fit. Three stages (NCC calculation, peak finding and subsample

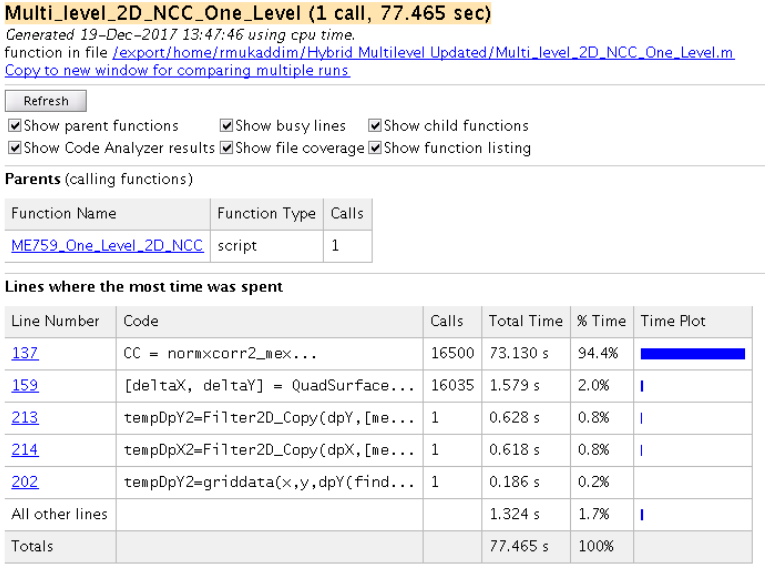


Figure 2 – MATLAB profiling result for sequential implementation of Multi-level displacement estimation.

estimation) are identified as computational bottleneck as shown in Figure 2. We attempted to perform these three stages in GPU to achieve speed up from the sequential implementation.

3.2.2 Displacement Estimation

NCC is calculated between 2-D windowed signals (kernel) between pre and post-compression frame. The typical kernel dimension in the axial direction, ker_y ranges from 100-300 pixels while the kernel dimension in lateral direction, ker_x ranges from 3-9 A-lines. Depending on kernel size and overlap among them, we can determine the size of displacement image defined as $[num_y \times num_x]$ with num_y is the number of sample points in axial direction while num_x is the number of sample points in lateral direction.

Figure 3 illustrates the process of estimation of one sample of the displacement image. Both pre and post compression images are divided into blocks based on user provided kernel size (black rectangular grid) and overlap. For this example, overlap is assumed to be zero and $[ker_y, ker_x] = [3,3]$. Displacements are estimated at the center point of each kernel. In the post-frame, a search region is established (blue rectangle) based on the search range parameter $[search_y, search_x]$ which in this case is [3, 3]. Kernel from pre-frame is matched with similar-sized kernels at each shift locations, $[search_y, search_x]$ in post-frame image using NCC resulting into NCC matrix shown as surface plot in Figure 3. Then displacement is estimated is though integer and subsample estimation of peak of the NCC function. The same process is repeated over all $[num_y \times num_x]$ points. The repeated computation nature of the problem makes it an ideal candidate for leveraging the advantage of GPU programing.

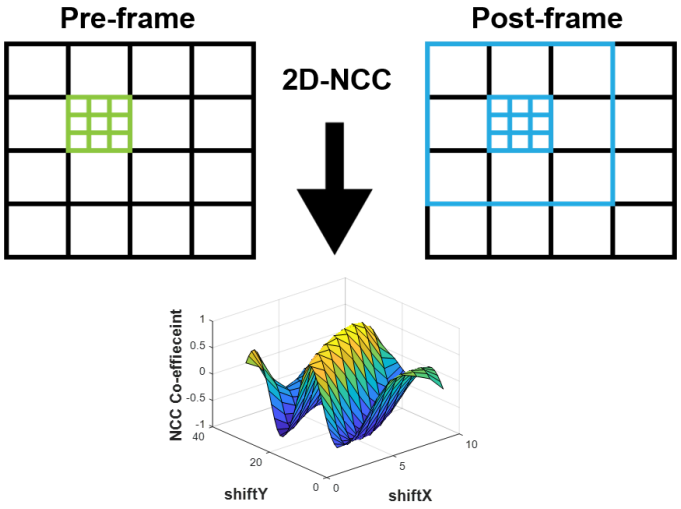


Figure 3 - NCC calculation criteria for one sample of displacement image.

3.2.3 Execution Model of Initial Implementation in GPU

In our execution model, each displacement point is calculated by one CUDA block resulting into a total $num_y \times num_x$ blocks. In each block, we have multiple threads calculating the NCC using (1) for each location. In particular, we have $search_y \times search_x$ number of threads for each shift location. Thus, displacement image samples translate into CUDA blocks and shifts for estimating each displacement points translate into CUDA threads. All together, we spawn $num_y \times num_x \times search_y \times search_x$ number of CUDA threads for one displacement image.

$$NCC(i,j) = \frac{\sum_{x=1}^{ker_y}\sum_{y=1}^{ker_x}\left[pre(x,y) - pre_{mean}\right]\left[post(x+i,y+i) - post_{mean}\right]}{\sqrt{\sum_{x=1}^{ker_y}\sum_{y=1}^{ker_x}\left[pre(x,y) - pre_{mean}\right]^2 \times \sum_{x=1}^{ker_y}\sum_{y=1}^{ker_x}\left[post(x+i,y+i) - post_{mean}\right]^2}} \quad (1)$$

$$i = 1, \ldots, search_y$$
$$j = 1, \ldots, search_x$$

3.3 MATLAB Wrapper for CUDA

The final implementation of this algorithm utilizes MATLAB with a wrapper function for the kernel call. Although a C implementation would be much faster than going through MATLAB, the nature of using this algorithm in the research setting with multiple different applications and datasets leans to the flexibility of MATLAB. We chose to maintain the similar MATLAB framework we have for the algorithm but work towards optimizing the bottlenecks of the program.

We settled to use MATLAB with a C++ wrapper. CUDA kernel written in C++ is compiled using NVCC to create object file using `'nvcc -O0 -std=c++11 –c normXcorr_Host_Mex_Final.cu –Xcompiler –fPIC -I/usr/local/MATLAB/ R2013a/extern/include;'`. Then, the mex function is written to create the appropriate input and output arrays for our CUDA implementation. Finally, we compile our mex file and link it to our CUDA object file using `'mex Corr2GPUMex_Final.cpp normXcorr_Host_Mex_Final.o –lcudart -L"/local/linux/cuda"'`. This process enables us to be in the MATLAB framework and at the same time make the most out of the hardware though CUDA programming.

3.4 Optimizations

The first implementation was programmed for code readability and to correctly match the execution model. No significant effort was made to optimize CUDA capabilities. This was done deliberately to get a baseline implementation and use the NVIDIA visual profiler to look for opportunities for analyze the code performance. The highest speedup was achieved after leveraging optimization opportunities, shown in Figure 5.

3.4.1 Thread Divergence

The initial implementation was analyzed using NVPP. The first optimization opportunity was inside the kernel where two conditional loops were causing ~7% thread divergence. In this context, each thread is performing NCC for a different subset of the sub-image. It is necessary to consider edge cases when there is only partial overlap between the pre and post sub-images. The initial implementation dealt with this using conditional statements inside a loop. The solution to eliminate thread divergence was to zero pad the images on the host side before passing the data to the device. Padding the input images yields an average speedup of 2.7x.

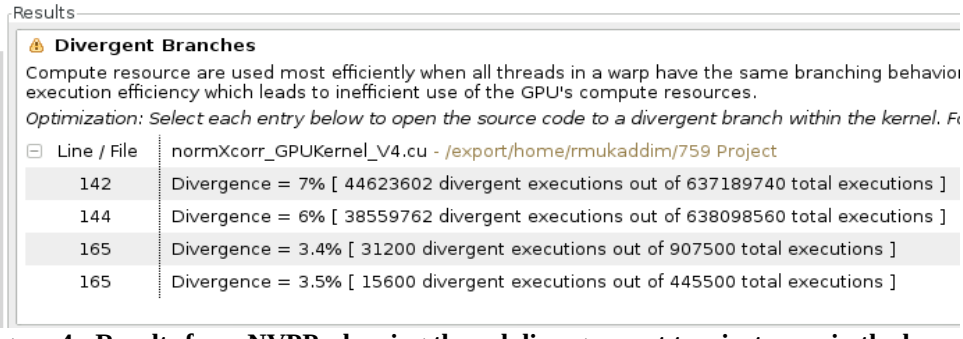


Figure 4 - Results from NVPP, showing thread divergence at two instances in the kernel.

3.4.2 Pinned Memory

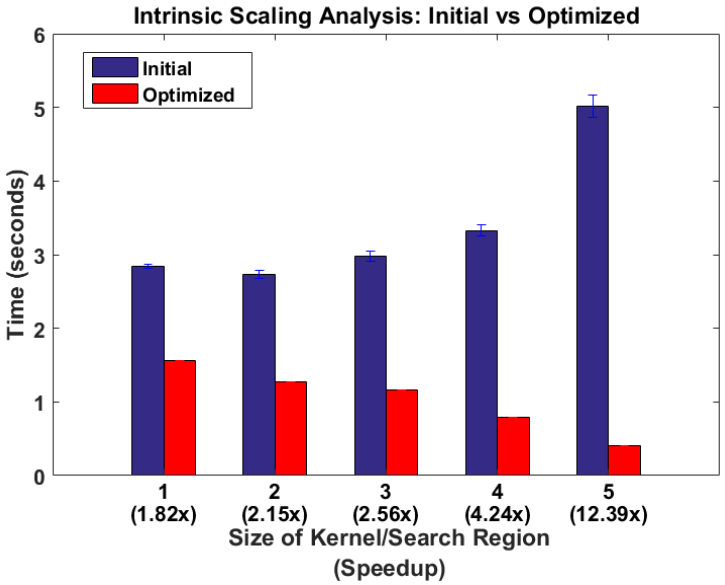
Data allocations on the CPU are pageable by default. The data can be temporarily page-locked to maximize the transfer speed between the host and device. Pinned memory also makes asynchronous memory transfers possible. We tested a version with pinned memory and asynchronous data transfers for the pre/post images. During testing, we saw no discernable difference in execution time. To investigate this, we found [9], where the behavior of one-way data transfers using pinned/non-pinned memory was investigated for varying amounts of data. In that specific study, the author concluded that pinned memory was only sensible if more than 128 MB of data was to be transferred. In this case, the author's conclusion agrees with our results since our image data is ~5 MB. We saw no significant speedup using pinned memory.

3.4.3 Shared Memory

The K40 GPU has 48 KB of shared memory and 16 streaming multiprocessors. Recall that each CUDA block is responsible for estimating the displacement of one image pixel by assigning each thread to perform the cross correlation for different locations inside the search region of a sub-image. Shared memory was used to take advantage of this data reuse and speedup the kernel execution. Each block loads its corresponding pre sub-image into shared memory for use in the CC calculation.

3.4.4 Peak Finding in GPU

The peak finding algorithm is used to find the peak of the correlation coefficient matrix along with the correlating x and y locations, which are consequently the x and y displacements for a given displacement pixel of the final displacement map. The peak finding algorithm uses reduction with a '$>$' operation. To obtain the x and y coordinates, the original index of each thread was moved along with the correlation coefficient movement. We chose to implement this in our own CUDA kernel, instead of implementing in thrust because the algorithm could then be optimized for this specific application. For performance of the final peak finding algorithm, the reduction operation used sequential addressing with loop unrolling and template parameters (similar to the reduce7 studied in class).



Bar Number	Kernel Y	Kernel X
1	161	9
2	129	9
3	97	9
4	65	9
5	33	9

Figure 5 – Improvements of performance after optimization. For variable kernel size, both baseline and optimized implementations were compared showing that the optimized implementation provides a greater speedup.

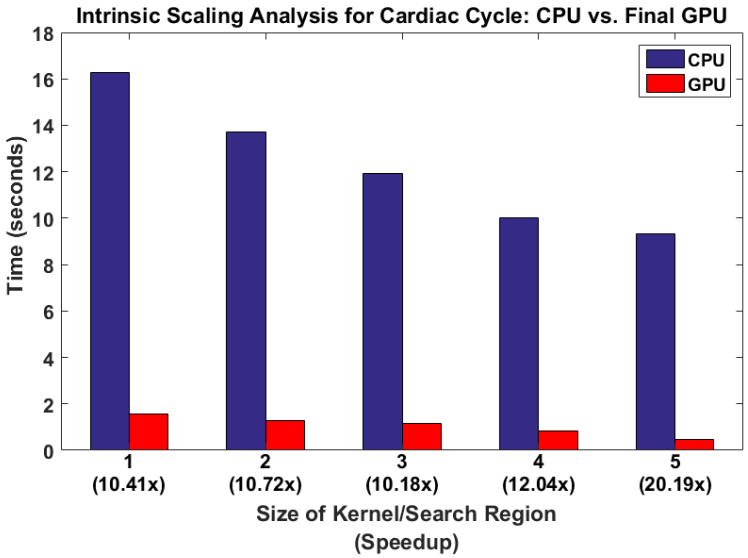
3.4.5 Subsample Displacement in GPU

Peak finding of correlation coefficient matrix provides integer displacement estimation. In order to estimate the subsample displacement, a 3×3-pixel region around maximum correlation peak is utilized to perform a 2D surface fit. This together with integer displacement provides the final displacement estimation. A kernel is written to perform 2D surface fit for each displacement point in parallel.

4. Results

4.1 Sequential vs Parallel Performance Comparison

To compare the difference between the CPU and GPU implementation, we designed a test that analyzed consecutive ultrasound images from a cardiac cycle (32 images total). The displacement estimation was done for 5 different kernel sizes to change the problem size. A comparison below shows how the execution time changes with problem size. The largest speedup (20.19x) was achieved in last test with the smallest kernel size.



Bar Number	Kernel Y	Kernel X
1	161	9
2	129	9
3	97	9
4	65	9
5	33	9

Figure 6 – Performance between sequential and parallel implementation

4.2 Exploring Performance and Accuracy through Applications in Liver

For liver ablation cases, the main purpose of displacement tracking is to better visualize the ablation zone. This means that the main location of interest is the actual ablation zone and it respecting "normal" liver on either side of the ablation zone at the same depth. More information can be found in [10, 11]. Some good measurements to quantify the visualization of the ablation zones are those of SNR and CNR shown in (2) and (3), respectfully.

$$SNR_e = \frac{\mu_s}{\sigma_s} \tag{2}$$

$$CNR = \frac{2(\mu_{s_1} - \mu_{s_2})^2}{(\sigma_{s_1}^2 + \sigma_{s_2}^2)} \tag{3}$$

Where μ and σ are the mean and standard deviation for s_1 and s_2 displacement values inside and outside the ablation zone.

Using this knowledge, the GPU algorithm for displacement estimation of liver ablation zones was compared to the CPU algorithm by utilizing the SNR and CNR. This was tested using 8 datasets, the results are shown in Figure 7 along with the resulting timings in Figure 8. As a note, Axial refers to the Y displacements and Lateral refers to the X displacements. The conclusion of these results shows that we obtain comparable results with a lower execution time.

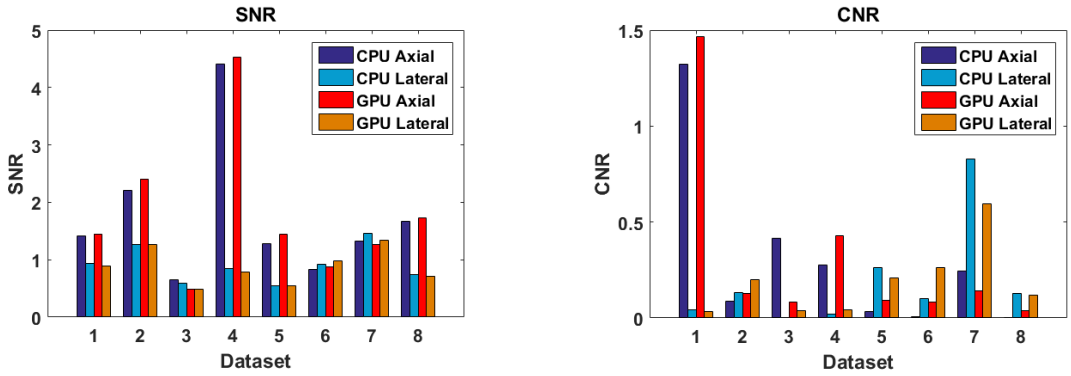


Figure 7 – The above bar plots show the SNR and CNR of the CPU vs GPU Multilevel tracking algorithm. The GPU algorithm is either the same or slightly greater than the CPU displacement data. This quantitatively shows that the GPU algorithm provides comparable results for displacement tracking of liver ablation zones as the CPU implementation.

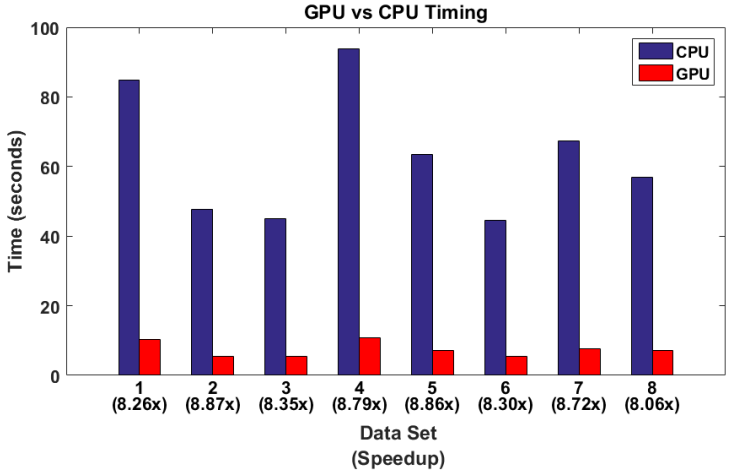


Figure 8 – This figure displays the execution time for both the CPU and GPU implementations of the Multilevel algorithm along with the associated speed up for each dataset.

5. Conclusions

We were successful in implementing the Multilevel Method on the GPU and comparing the results to the previous CPU implementation. The GPU showed similar results as the CPU implementation with speedups of 8-20x.The summary of speedups that are seen from the GPU implementation are shown in Table 2.

Table 2 - The summary of speedups seen for various kernel and search sizes.

Kernel Size [x, y]	Search Size [x, y]	Average Speedup
[7, 171]	[5, 41]	8.53x
[9, 161]	[17, 49]	10.41x
[9, 129]	[17, 39]	10.72x
[9, 97]	[17, 31]	10.18x
[9, 65]	[17, 21]	12.04x
[9, 33]	[17, 11]	20.19x

In future work, we will translate the algorithm to work with more levels to get higher displacement tracking accuracy. Further study will go into the effects of the different displacements seen between the CPU and GPU implementations, and whether or not they will harm performance as more levels are added.

6. References

[1] H. R. Shi and T. Varghese, "Two-dimensional multi-level strain estimation for discontinuous tissue," (in English), *Physics in Medicine and Biology*, vol. 52, no. 2, pp. 389-401, Jan 21 2007.

[2] S. Deka, "A CPU-GPU Hybrid Approach for Accelerating Cross-correlation Based Strain Elastography," Texas A & M University, 2011.

[3] N. Deshmukh, H. Rivaz, and E. Boctor, "GPU-based elasticity imaging algorithms," in *Proc. Int. Conf. Med. Imag. Comp. & Comp. Assist. Interven*, 2009.

[4] N. H. Meshram and T. Varghese, "Fast multilevel Lagrangian carotid strain imaging with GPU computing," in *Ultrasonics Symposium (IUS), 2017 IEEE International*, 2017, pp. 1-4: IEEE.

[5] E. Montagnon, S. Hissoiny, P. Després, and G. Cloutier, "Real-time processing in dynamic ultrasound elastography: A GPU-based implementation using CUDA," in *Information Science, Signal Processing and their Applications (ISSPA), 2012 11th International Conference on*, 2012, pp. 472-477: IEEE.

[6] C. Nvidia, "Nvidias next generation cuda compute architecture: Kepler gk110," *Whitepaper*, 2012.

[7] J. Ophir, F. Kallel, T. Varghese, M. Bertrand, I. Cespedes, and H. Ponnekanti, "Elastography: A systems approach," *International journal of imaging systems and technology*, vol. 8, no. 1, pp. 89-103, 1997.

[8] F. Kallel and J. Ophir, "A least-squares strain estimator for elastography," *Ultrasonic imaging*, vol. 19, no. 3, pp. 195-208, 1997.

[9] M. Boyer. *Pinned vs. Non-pinned Memory*. Available: https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html

[10] S. Bharat, T. Varghese, E. L. Madsen, and J. A. Zagzebski, "Radio-frequency ablation electrode displacement elastography: a phantom study," *Med Phys*, vol. 35, no. 6, pp. 2432-42, Jun 2008.

[11] N. Rubert *et al.*, "Electrode displacement strain imaging of thermally-ablated liver tissue in an in vivo animal model," *Med Phys*, vol. 37, no. 3, pp. 1075-82, Mar 2010.