Program Digest, 2019 IEEE International Ultrasonics Symposium (IUS)
Glasgow, Scotland, October 6-9, 2019

A GPU-Based Implementation of ADMIRE

Christopher Khan
Department of Biomedical Engineering
Vanderbilt University
Nashville, TN
christopher.m.khan@vanderbilt.edu

Kazuyuki Dei
Department of Biomedical Engineering
Vanderbilt University
Nashville, TN
kazuyuki.dei@vanderbilt.edu

Brett Byram
Department of Biomedical Engineering
Vanderbilt University
Nashville, TN
brett.c.byram@vanderbilt.edu

***Abstract*—Multipath and off-axis scattering are two of the primary mechanisms for ultrasound image degradation. To address their impact, we have proposed Aperture Domain Model Image REconstruction (ADMIRE). This algorithm utilizes a model-based approach in order to identify and suppress sources of acoustic clutter. The ability of ADMIRE to suppress clutter and improve image quality has been demonstrated in previous works, but its use for real-time imaging has been infeasible due to its significant computational requirements. However, in recent years, the use of GPUs for general-purpose computing has enabled significant acceleration of compute-intensive algorithms. This is due to the fact that many modern GPUs have thousands of computational cores that can be utilized to perform massively parallel processing.**

Therefore, in this work, we have developed a GPU-based implementation of ADMIRE. The computations were distributed across two GPUs, and speedups of almost three orders of magnitude were achieved when compared to a serial CPU implementation. The frame rate depends upon various imaging parameters, and we demonstrate this using a small cyst simulation dataset and a large in-vivo kidney dataset. However, even for the large dataset, the implementation still provides the ability to process multiple frames of data per second. Due to this, it has the capability to serve as a real-time imaging framework.

***Index Terms*—Ultrasound, GPU computing, real-time imaging**

I. Introduction

One of the fundamental advantages of using ultrasound as a medical imaging modality is its ability to provide real-time imaging capabilities. Due to this, the most commonly used ultrasound beamforming method today is delay-and-sum (DAS) beamforming. This method is simple in that it consists of only two steps. The first step is to time-delay the ultrasound channel data in order to adjust for path length differences between the transducer elements and the returning acoustic wavefronts, and the second step is to coherently sum the received signals across the aperture in order to obtain RF data. The simplicity of the pipeline has led to real-time implementations of the method being deployed on clinical scanners. Although DAS is widely used and has been implemented in real-time, it still has important disadvantages. One of the most important disadvantages is that it is less effective than advanced beamforming methods when it comes to addressing mechanisms such as multipath and off-axis scattering, which produce acoustic clutter that degrades image quality [1].

To address these mechanisms, we have previously proposed Aperture Domain Model Image REconstruction (ADMIRE) [2], [3], [4]. The basis of this method is that it uses a model-based approach to fit the aperture domain data and reconstruct channel data that is decluttered. In particular, the Short-Time Fourier Transform (STFT) of the time-delayed ultrasound channel data is taken, and the aperture domain data for several frequencies within each STFT window is fit using models. The model matrix for each frequency consists of a grid of scattering locations that can contribute to the observed aperture domain signal, and each predictor represents the received aperture signal for a wavefront, localized in time and frequency, that is returning from one scattering location. A linear regression model with elastic-net regularization is used to determine the contributions of these scattering locations, and the decluttered signal can then be reconstructed by only utilizing the locations that do not contribute to multipath or off-axis scattering. The decluttered channel data is obtained by taking the Inverse Short-Time Fourier Transform (ISTFT).

Although ADMIRE can improve image quality by suppressing acoustic clutter, it is computationally demanding. In particular, the model decomposition stage of the pipeline is the primary bottleneck because it typically requires thousands of individual model fits to be performed. To reduce the computing time required for these fits, a computationally-efficient implementation of ADMIRE was previously developed [5]. This implementation utilizes independent component analysis (ICA) in order to reduce the model matrix size for each fit. For example, without ICA, the size of each model matrix X is $X \in \mathbb{C}^{\mathbb{M}\times\mathbb{N}}$, where $\mathbb{M}$ is the number of aperture elements and $\mathbb{N}$ is the number of model predictors. With ICA, the size is $X \in \mathbb{C}^{\mathbb{M}\times 2\mathbb{M}}$, which is much smaller because $\mathbb{M} < \mathbb{N}$. For example, a typical value for $\mathbb{M}$ might be 128, while a typical value for $\mathbb{N}$ might range from 10,000 to 1,000,000. The second dimension of the matrix is $2\mathbb{M}$ because ICA is applied individually to the group of predictors that are within the desired signal ROI and the group of predictors that are not within the ROI. These two matrices are then concatenated together. This reduced model significantly decreases ADMIRE's computational time, but it still requires several minutes to process one frame of channel data in many cases.

Real-time image processing with ADMIRE is currently infeasible on a CPU. However, by using graphics processing units (GPUs), the computational speed can be dramatically improved. This is because GPUs are designed for massively parallel processing, and the entire ADMIRE pipeline can be

978-1-7281-4595-2/19/$31.00 ©2019 IEEE

executed in parallel. Real-time GPU implementations of other compute-intensive beamforming algorithms have already been developed. For example, Hyun et al. developed a real-time GPU implementation of short lag spatial coherence imaging (SLSC) [6], [7], and Chen et al. developed a real-time GPU implementation of minimum variance beamforming [8]. Therefore, in this work, we have developed a GPU implementation of ADMIRE. Moreover, we demonstrate the ability of the implementation to perform real-time image processing.

II. Methods

To develop the GPU implementation of ADMIRE, the C programming language was utilized along with Nvidia's Compute Unified Device Architecture (CUDA) parallel programming platform. The basis of this framework is that data is transferred from host memory to the GPU's memory, where CUDA kernels are called for parallel processing. The processed data is then transferred back to host memory. In this case, the undelayed ultrasound channel data along with various imaging parameters is being transferred to the GPU.

Once on the GPU, a CUDA kernel is utilized to compute the delays for the data. In terms of the parallelism of the GPU, it consists of streaming multiprocessors, and each one contains many computational cores. Therefore, in a kernel, blocks of GPU threads are initialized and distributed across these streaming multiprocessors, where each thread is executed on a computational core. For the delay kernel, each initialized block corresponds to a specific depth and beam position (assuming focused transmits), and each thread within a block calculates the delay for an individual element. These delays are then applied to the channel data in a separate kernel. This kernel initializes the same number of blocks and threads as the previous one, but each thread is now being used to perform linear interpolation. The undelayed channel data is bound to texture memory upon transfer to the GPU, and this type of memory allows for fast interpolation to be performed. Each thread performs a fetch from texture memory in order to obtain its corresponding time-delayed data value.

After the channel data is delayed, the Short-Time Fourier Transform (STFT) is taken with 0% window overlap. This involves first using a kernel in order to obtain the data for each STFT window. The kernel initializes blocks that each correspond to a specific element, beam, and set of windows. Within a single block, each thread corresponds to an individual sample within one of the windows for the window group. For example, if a single STFT window contains 10 data samples with 10 additional zeros for zero padding, then 20 threads will be utilized for that particular window. 10 threads will fetch the 10 data samples, and the other 10 threads will store zeros in the respective window positions. In addition, the coefficients for any windowing function can be transferred to the GPU and multiplied to the data samples in order to perform windowing.

Once the data for each STFT window has been arranged, the Nvidia cufft library is utilized to perform an in-place, batched complex to complex transform. This involves taking a 1D Fourier Transform along the depth dimension for each element of each window. Due to the fact that a complex to complex transform is used, zeros are stored for the imaginary component of the input data. Following the batched transform, each window contains the aperture data for multiple frequencies. In ADMIRE, a subset of the frequencies corresponding to the pulse bandwidth are typically fit. Therefore, a kernel is utilized to obtain the data that only corresponds to the frequencies that are selected for model decomposition. Each block in the kernel corresponds to a specific STFT window and beam, and each thread corresponds to a specific element. Every thread loops through all of the desired frequencies within a window and obtains the data for the element that the thread corresponds to. The data is stored into a new array in such a format that successive sets of aperture data follow one another. Moreover, each set of aperture data has all of the real components stored first followed by the imaginary components.

To prepare for model decomposition, another kernel is utilized. Each thread in the kernel handles a specific set of aperture data, and as many blocks are initialized as are required for all aperture datasets to be accounted for. Each thread first accounts for aperture growth if it is applied. To do so, an array containing binary masks for the aperture data is transferred to the GPU. If the binary mask for a specific aperture data set contains a 0 in a certain position, then the thread will not include the data in that position. However, if there is a 1, then the thread will include the data in that position. The data with aperture growth accounted for is stored into a new array that contains successive sets of aperture data. Each thread also standardizes its set of aperture data within this new array. This involves dividing each set of aperture data by its standard deviation ($\frac{1}{N}$ formula). In addition, the λ parameter that is used when performing linear regression with elastic-net regularization is calculated for each set of aperture data. This is calculated as $0.0189\sqrt{\frac{\boldsymbol{y}^T\boldsymbol{y}}{N}}$, where $\mathbf{y}$ is one set of aperture data before being standardized. To account for the standardization, each λ value is also divided by its respective aperture dataset standard deviation. The standard deviation values and λ values are stored into separate arrays on the GPU.

As previously stated, linear regression with elastic-net regularization is utilized to fit each predictor matrix to its corresponding set of aperture data. In the CPU implementation of ADMIRE, the glmnet software package [9] is used to perform the fits in a serial fashion. However, in the GPU implementation, thousands of these fits can be performed simultaneously. This is due to the fact that we have developed a custom GPU implementation of cyclic coordinate descent, which is the optimization method that is used to determine the model coefficients for each fit. This method is also used by glmnet, and it involves minimizing the objective function shown in (1).

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\arg\min} \left(\frac{1}{2N} \sum_{i=1}^{N} \left(y_i - \boldsymbol{x}_i^T \boldsymbol{\beta} \right)^2 + \lambda \left(\alpha \left\| \boldsymbol{\beta} \right\|_1 + \frac{(1-\alpha) \left\| \boldsymbol{\beta} \right\|_2^2}{2} \right) \right) \tag{1}$$

Cyclic coordinate descent is executed on the GPU by

having each thread perform a fit involving a specific set of aperture data. As many blocks are initialized as are required for handling all of the sets of aperture data corresponding to the different frequencies, STFT windows, and beams. All of the predictor matrices can be pre-computed and transferred to the GPU only once because the same models can be reused for different image frames. These predictor matrices are all collapsed and stored into a 1D array before they are transferred. The array strides for where the data for each fit begins are also transferred to the GPU due to the fact that aperture growth causes many of the fits to have a different number of aperture elements. Each thread performs cyclic coordinate descent until one of two convergence criteria is first met. These two convergence criteria include setting a limit to the maximum number of iterations allowed and setting a tolerance for the maximum model coefficent change between iterations. These convergence criteria are different from glmnet, which utilizes a criterion that focuses on the impact of the change in coefficients on a fit as described in [9]. Once the optimization is terminated, each thread unstandardizes the model coefficients for the fit by multiplying them by the standard deviation that was previously stored. Each thread then reconstructs its decluttered aperture data by only using the predictors corresponding to a specific ROI as shown in (2). By doing so, the contributions of the predictors that correspond to off-axis scattering and multipath scattering are eliminated.

$$\boldsymbol{y}_{ROI} = \boldsymbol{X}_{ROI}\hat{\boldsymbol{\beta}}_{ROI} \tag{2}$$

Once reconstruction is complete, another kernel is utilized to store the data back into the original array containing all of the STFT data for all of the frequencies. The frequencies for which a model fit is not performed are zeroed out. The number of blocks initialized for the kernel corresponds to the number of STFT windows, and each thread within a block stores the reconstructed aperture data for all of the fitted frequency cases within the STFT window for a specific beam. The negative frequencies are not fitted in order to save computational time. Therefore, for each positive frequency that was fitted, the complex conjugate of its reconstructed signal data is stored for the corresponding negative frequency. After the data is stored, the cufft library is used to perform an in-place, batched complex to complex transform. This process involves taking a 1D Inverse Fourier Transform along the depth dimension for each element of each window. As previously stated, the window overlap used for the STFT is 0%, so performing just the 1D Inverse Fourier Transforms gives the Inverse Short Time Fourier Transform of the data in this case.

A kernel following the Inverse Short Time Fourier Transform is then used to obtain the decluttered channel data with the samples corresponding to the zero padded positions removed. Each block corresponds to a specific depth region and beam, and each thread obtains the depth samples within the depth region for a specific element. Only the real component of the data is stored for each sample, and the data is also normalized by the Inverse Fourier Transform length. To obtain the summed RF data from the channel data, a summing kernel is used. Each block corresponds to a specific depth and beam, and each thread corresponds to a specific element. The sum is computed within each block, and this represents the coherently summed aperture data for each image location. An optimized summing algorithm as described in [10] is used whenever possible. The envelope data is calculated from the RF data by first taking the Hilbert Transform. The process for taking the Hilbert Transform follows the discrete algorithm that is described in [11] and is used by MATLAB. On the GPU, this involves performing an in-place, batched complex to complex transform using the cufft library in order to compute the 1D Fourier Transform of each beam. A kernel is then used to weight the different frequency bins. Each block in the kernel corresponds to a specific frequency bin, and each thread corresponds to a specific beam. Once the data is weighted, the cufft library is used to perform an in-place, batched complex to complex transform, which gives the 1D Inverse Fourier Transform of each beam.

The magnitude of the IQ data is computed using a separate kernel. Each block corresponds to a specific depth, and each thread corresponds to a specific beam. A single thread takes the complex data sample for its depth and beam position, and it computes its magnitude after normalizing it by the Inverse Fourier Transform length. The results are stored into a separate array, which is then transferred back to the host. The envelope is log compressed and displayed within MATLAB.

III. Results

To compare the GPU and CPU implementations of ADMIRE, benchmarks were performed utilizing an in-vivo kidney dataset and a Field II cyst simulation dataset. The model matrices for both cases were pre-computed, and ICA was applied. The computations for the GPU implementation were split across a GeForce GTX 1080 Ti GPU and a GeForce RTX 2080 Ti GPU. More beams were distributed to the 2080 Ti GPU due to the fact that it is more powerful. The host computer used for the benchmarks contained dual Intel Xeon Silver 4114 CPUs @ 2.20 GHz with 10 cores each. The MATLAB programming language was used for the CPU implementation, and the number of computational threads was set to 1. To perform cyclic coordinate descent for the CPU implementation, a MEX file written in Fortran from the glmnet software package was utilized. The imaging/processing parameters can be seen in Table I. Fig. 1 shows the in-vivo kidney and simulated cyst datasets processed with DAS, GPU ADMIRE, and CPU ADMIRE. The processing times are displayed on each image, and the contrast ratio values are also shown for a given ROI. Both of the datasets were already time-delayed, so this was not included in the run times. Moreover, post-envelope calculation steps such as normalization, log compression, and scan conversion (kidney dataset) were not included in the timing either. Data transfer times between host and GPU memory were included in the GPU implementation run times.

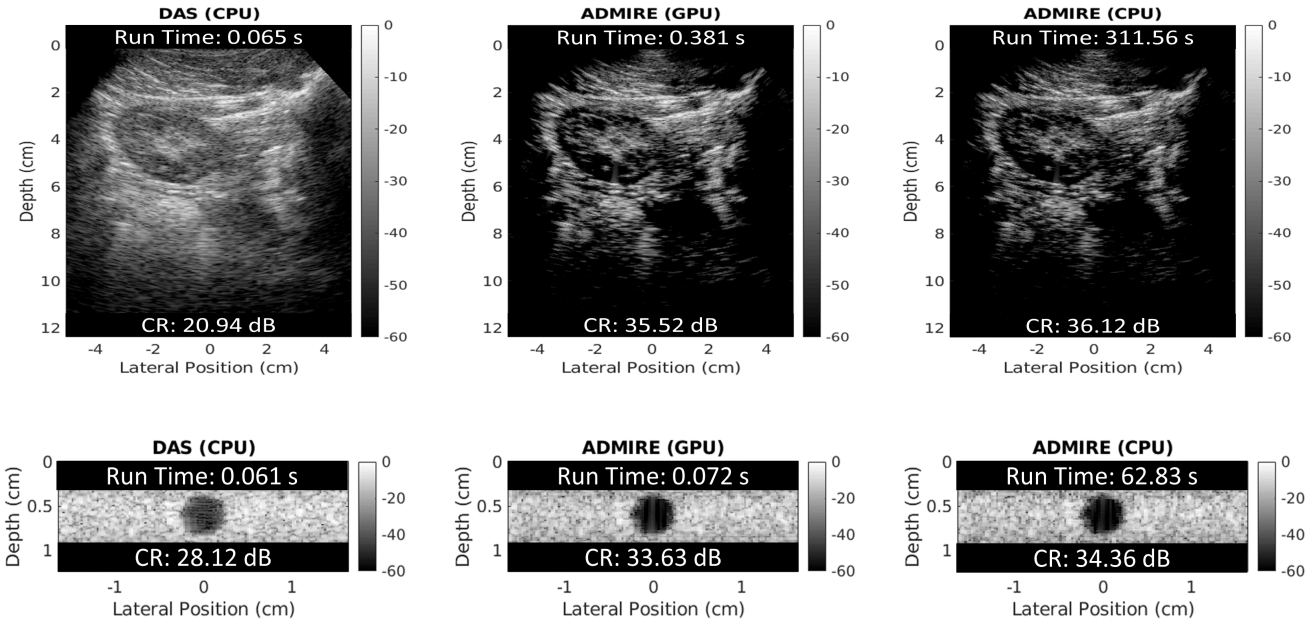

Fig. 1. Images processed with DAS (left), GPU ADMIRE (center), and CPU ADMIRE (right). The top row is for the in-vivo kidney dataset, and the bottom row is for the simulated cyst dataset.

TABLE I: Imaging/Processing Parameters for Each Dataset		
Imaging/Processing Parameter	**Kidney Dataset**	**Cyst Dataset**
Depth Samples	2,235	640
Elements	128	128
Beams	121	128
f0 (MHz)	3.47	3
fs (MHz)	13.89	40
Padded STFT Window Length	10	40
STFT Window Overlap	0%	0%
Frequencies Fit per STFT Window	3	4
α for Regularization	0.9	0.9
Coord. Descent Max Iterations	100,000	100,000
Coord. Descent Coeff. Change Tol.	10	10

IV. Discussion

As shown in Fig. 1, the GPU implementation of ADMIRE provides a computational speedup of almost three orders of magnitude when compared to the CPU implementation while still providing a similar increase in contrast ratio. ADMIRE processing on a GPU took only 72 ms for the cyst dataset, and even for the kidney dataset, processing took 381 ms. This means that real-time imaging can be achieved for many applications. Frame rates can be further increased by taking advantage of the fact that the entire field-of-view does not necessarily need to be processed with ADMIRE. For example, in a procedure such as a biopsy, ADMIRE can be applied to an ROI corresponding to the area of tissue removal while DAS can applied to the other regions that are of lesser interest.

V. Conclusions

We have developed a GPU implementation of ADMIRE that is almost three orders of magnitude faster than the CPU implementation. Moreover, we have demonstrated the potential of this implementation to be used for real-time imaging. Future work includes further optimizing the code and interfacing it with a Verasonics Vantage 128 ultrasound research system.

VI. Acknowledgement

This work was supported by NIH grants R01EB020040 and S10OD016216-01 and NAVSEA grant N0002419C4302.

References

[1] G. Pinton, G. Trahey, and J. Dahl. "Sources of image degradation in fundamental and harmonic ultrasound imaging using nonlinear, full-wave simulations." *IEEE transactions on ultrasonics, ferroelectrics, and frequency control* 58.4 (2011): 754-765.

[2] B. Byram and M. Jakovljevic. "Ultrasonic multipath and beamforming clutter reduction: a chirp model approach." *IEEE transactions on ultrasonics, ferroelectrics, and frequency control* 61.3 (2014): 428-440.

[3] B. Byram, K. Dei, J. Tierney, and D. Dumont. "A model and regularization scheme for ultrasonic beamforming clutter reduction." *IEEE transactions on ultrasonics, ferroelectrics, and frequency control* 62.11 (2015): 1913-1927.

[4] K. Dei and B. Byram. "The impact of model-based clutter suppressionon cluttered, aberrated wavefronts," *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, vol. 64, no. 10, pp. 1450–1464, 2017.

[5] K. Dei, S. Schlunk, and B. Byram. "Computationally-Efficient Implementation of Aperture Domain Model Image Reconstruction (ADMIRE)." *IEEE transactions on ultrasonics, ferroelectrics, and frequency control* (2019).

[6] D. Hyun, G. Trahey, and J. Dahl. "In vivo demonstration of a real-time simultaneous B-mode/spatial coherence GPU-based beamformer." *2013 IEEE International Ultrasonics Symposium (IUS)*. IEEE, 2013.

[7] D. Hyun, G. Trahey, and J. Dahl. "Real-time high-framerate in vivo cardiac SLSC imaging with a GPU-based beamformer." *2015 IEEE International Ultrasonics Symposium (IUS)*. IEEE, 2015.

[8] J. Chen, B. Yiu, H. So, and A. Yu. "Real-time GPU-based adaptive beamformer for high quality ultrasound imaging." *2011 IEEE International Ultrasonics Symposium*. IEEE, 2011.

[9] T. Hastie and J. Qian. "Glmnet vignette." *Retrieve from http://www. web. stanford. edu/˜ hastie/Papers/Glmnet_Vignette. pdf. Accessed September* 20 (2014): 2016.

[10] M. Harris. "Optimizing parallel reduction in CUDA." *Nvidia developer technology* 2.4 (2007): 70.

[11] L. Marple. "Computing the Discrete-Time Analytic Signal via FFT." *IEEE transactions on signal processing*. vol. 47, 1999, pp. 2600–2603.