A Real-Time, GPU-Based Implementation of Aperture Domain Model Image REconstruction

Christopher Khan, *Student Member, IEEE*, Kazuyuki Dei, Siegfried Schlunk, *Student Member, IEEE*, Kathryn Ozgun, *Student Member, IEEE*, and Brett Byram, *Member, IEEE*

***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 graphics processing units (GPUs) for general-purpose computing has enabled the significant acceleration of compute-intensive algorithms. This is because 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 implementation on a single GPU provides a speedup of two orders of magnitude when compared to a serial CPU implementation, and additional speedup is achieved when the computations are distributed across two GPUs. In addition, we demonstrate the feasibility of the GPU implementation to be used for real-time imaging by interfacing it with a Verasonics Vantage 128 ultrasound research system. Moreover, we show that other beamforming techniques, such as delay-and-sum (DAS) and short-lag spatial coherence (SLSC), can be computed and simultaneously displayed with ADMIRE. The frame rate depends upon various parameters, and this is exhibited in the multiple imaging cases that are presented. An open-source code repository containing CPU and GPU implementations of ADMIRE is also provided.**

***Index Terms*— Graphics processing unit (GPU) computing, real-time imaging, ultrasound.**

Manuscript received January 5, 2021; accepted January 27, 2021. Date of publication February 2, 2021; date of current version May 25, 2021. This work was supported in part by NIH under Grant R01EB020040 and Grant S10OD016216-01, in part by Naval Sea Systems Command (NAVSEA) under Grant N0002419C4302, and in part by NSF under Award IIS-1750994. *(Corresponding author: Christopher Khan.)*

Christopher Khan, Siegfried Schlunk, Kathryn Ozgun, and Brett Byram are with the Department of Biomedical Engineering, School of Engineering, Vanderbilt University, Nashville, TN 37235-1826 USA (e-mail: christopher.m.khan@vanderbilt.edu).

Kazuyuki Dei was with the Department of Biomedical Engineering, School of Engineering, Vanderbilt University, Nashville, TN 37235-1826 USA. He is now with GE Healthcare, Wauwatosa, WI 53226-4856 USA.

This article has supplementary downloadable material available at https://doi.org/10.1109/TUFFC.2021.3056334, provided by the authors.

Digital Object Identifier 10.1109/TUFFC.2021.3056334

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, delay-and-sum (DAS) beamforming is the primary ultrasound beamforming method that is used today. 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. 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 previously proposed Aperture Domain Model Image REconstruction (ADMIRE) [2]–[4]. The basis of this method is that it uses a model-based approach to fit the aperture domain data and reconstruct decluttered channel data. ADMIRE is able to achieve significant improvements in ultrasound image quality by suppressing acoustic clutter while still preserving fundamental B-mode image characteristics, but one primary barrier has prevented it from achieving clinical translation. This barrier is the high computational complexity of the algorithm, which has made it infeasible to implement it in real time on a CPU. Essentially, the ability of ADMIRE to provide image quality improvements is overshadowed by the inability of these improvements to be realized in real time, which means that it effectively cannot be used for its ultimate goal of improving diagnostic ultrasound examinations. However, by using graphics processing units (GPUs), the computational speed can be dramatically improved in order to overcome this barrier and allow for potential widespread adoption. This is because GPUs are designed for massively parallel processing, and the entire ADMIRE pipeline can be executed in parallel. Moreover, unlike field-programmable gate arrays (FPGAs), a traditional software programming approach can be used with GPUs rather than a hardware approach, which typically allows for reduced development time and costs. This also allows for potential

future algorithmic modifications to be easily made in software rather than having to modify and perform verification of a hardware design. In addition, another benefit of using GPUs is that a better price efficiency can be achieved when compared to FPGAs in terms of the monetary cost per GFLOP of computational performance [5].

Real-time GPU implementations of other compute-intensive beamforming algorithms have already been developed. For example, short-lag spatial coherence (SLSC) beamforming involves calculating the spatial coherence of backscattered echoes, and it assumes that mechanisms of image degradation contribute to signal incoherence [6]. This method requires a large number of computations to be performed due to the fact that for a given pixel, the coherence across the aperture must be calculated. Despite this, these computations can be performed in parallel in order to achieve significant speedup. Therefore, Hyun *et al.* [7]–[9] developed a GPU implementation of this technique that was able to achieve real-time imaging. Minimum variance beamforming is another technique that is compute-intensive. This technique improves image quality by applying an adaptive weighting scheme to the received data in order to position side lobes in directions where there is a low amount of received energy, but obtaining these adaptive weights requires the computation of covariance matrices and their inverses [10], [11]. However, like the coherence computations for SLSC, these computations can also be performed in parallel. Due to this, real-time GPU implementations of this technique have been developed [12], [13].

Aside from the aforementioned examples, GPUs have also been used in additional ultrasound applications, such as plane-wave compounding and synthetic aperture imaging [14], displacement estimation [15], volumetric image reconstruction [16], clutter filtering [17], and speckle reduction [18]. This is significant due to the fact that it highlights a movement toward the adoption of GPU-based processing in the field of ultrasound. Furthermore, the number of applications that utilize GPUs will likely increase at an even faster rate as tools such as open-source GPU processing frameworks [19], [20] continue to be made available. Therefore, GPU computing can be utilized as a powerful tool for ultrasound, and in this work, we have leveraged this tool to develop a GPU-based implementation of ADMIRE. By developing this implementation, we have demonstrated that the computational run-time burden of ADMIRE can be overcome in order to make clinical translation feasible.

II. Methods

A. Overview of ADMIRE

ADMIRE utilizes a model-based approach in order to reconstruct decluttered channel data. In particular, the short-time Fourier transform (STFT) of the time-delayed ultrasound channel data is calculated, 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. The benefit of using elastic-net regularization is that it provides a weighting between L1-regularization and L2-regularization, which results in variable selection and coefficient shrinkage being performed while still allowing for groups of correlated predictors to be present in a model [21]. Once the model fits are performed, 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). An overview of ADMIRE is provided in Fig. 1.

Regarding computational time, the model fitting and reconstruction 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 [22]. This implementation utilizes fourth-order blind identification independent component analysis (FOBI-ICA) [23], [24] in order to reduce the model matrix size for each fit while still preserving image quality [22]. Without ICA, the size of each model matrix $\boldsymbol{X}$ is $\boldsymbol{X} \in \mathbb{C}^{\mathbb{M}\times\mathbb{P}}$, where $\mathbb{M}$ is the number of aperture elements and $\mathbb{P}$ is the number of model predictors. With ICA, the size is $\boldsymbol{X} \in \mathbb{C}^{\mathbb{M}\times 2\mathbb{M}}$, which is much smaller due to the fact that $\mathbb{M} < \mathbb{P}$. A typical value for $\mathbb{M}$ might be 128 while a typical value for $\mathbb{P}$ 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 region of interest (ROI) and the group of predictors that are not within the ROI. These two matrices are then concatenated together, and the real and imaginary components are tiled as shown below such that the matrix size is $\boldsymbol{X} \in \mathbb{R}^{2\mathbb{M}\times 4\mathbb{M}}$

$$\boldsymbol{X} = \begin{bmatrix} \Re\{\boldsymbol{X}_{\text{ROI}}\} & \Re\{\boldsymbol{X}_{\text{Outer}}\} & -\Im\{\boldsymbol{X}_{\text{ROI}}\} & -\Im\{\boldsymbol{X}_{\text{Outer}}\} \\ \Im\{\boldsymbol{X}_{\text{ROI}}\} & \Im\{\boldsymbol{X}_{\text{Outer}}\} & \Re\{\boldsymbol{X}_{\text{ROI}}\} & \Re\{\boldsymbol{X}_{\text{Outer}}\} \end{bmatrix}.$$

This reduced model significantly decreases ADMIRE's computational time, but real-time imaging is still infeasible with the CPU implementation.

B. Overview of GPU-Based Processing Pipeline

In order to assist others in creating novel GPU implementations of ultrasound beamforming algorithms, we include significant details about our processing pipeline, and we have made our code available in an open-source code repository. To develop the GPU implementation of ADMIRE, the C programming language was utilized along with NVIDIA's (NVIDIA Corporation, Santa Clara, CA, USA) Compute Unified Device Architecture (CUDA) parallel programming platform. The MATLAB (The MathWorks, Inc., Natick, MA, USA) programming language was also utilized in order to allow the C/CUDA code to be called within MATLAB through a MEX-interface. 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. The entire processing pipeline is shown in Fig. 2. The following

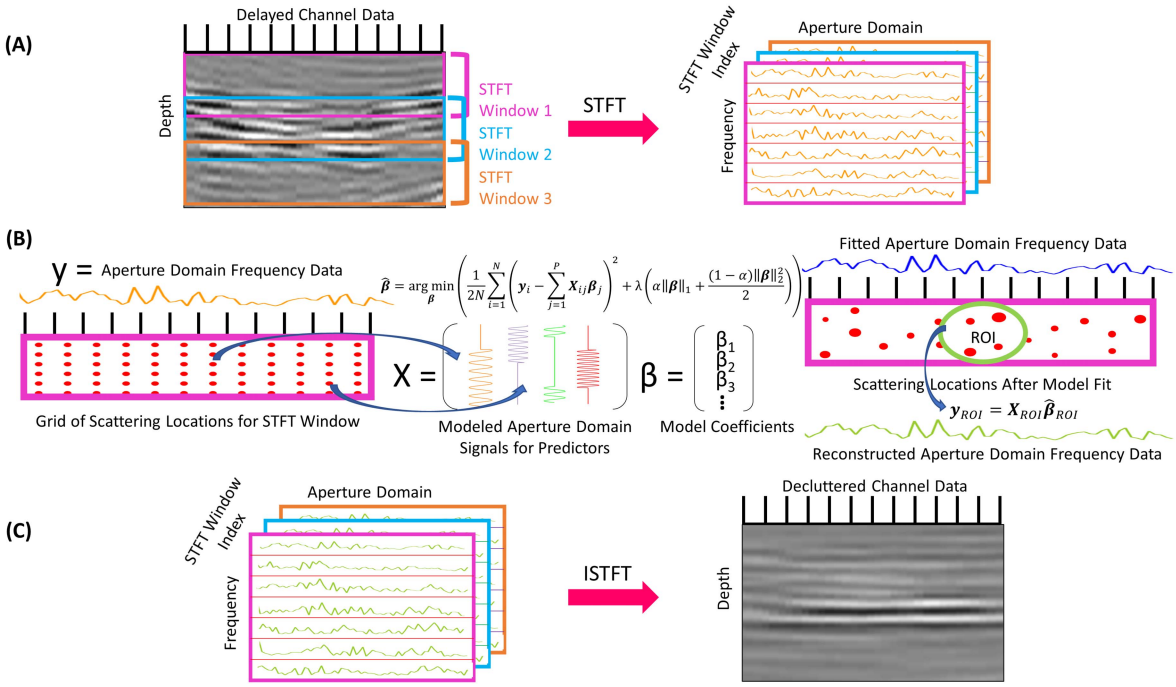

Fig. 1. Overview of ADMIRE. (A) Obtain the time-delayed channel data and calculate the STFT along the depth dimension for each channel. (B) Obtain the corresponding model matrix for each set of aperture domain frequency data that will be reconstructed in each STFT window in each beam, fit each model matrix to its corresponding set of aperture domain frequency data (sizes of red points correspond to how much each scattering location contributes to the aperture domain frequency data), and reconstruct each set of aperture domain frequency data by only using the predictors that correspond to scattering locations that are within an ROI. (C) Calculate the ISTFT of the reconstructed aperture domain frequency data in order to obtain the decluttered channel data. Note that the scattering locations are not restricted to the depth range of the STFT window. The grid of scattering locations illustrated in (B) corresponds to the first STFT window. For STFT windows that correspond to deeper depths, the scattering locations can also be located in shallower depths because these locations can contribute to off-axis scattering and multipath scattering that affect the aperture domain frequency data for the STFT window. Essentially, as the depths become deeper for subsequent STFT windows, the depth range for possible scattering locations also increases.

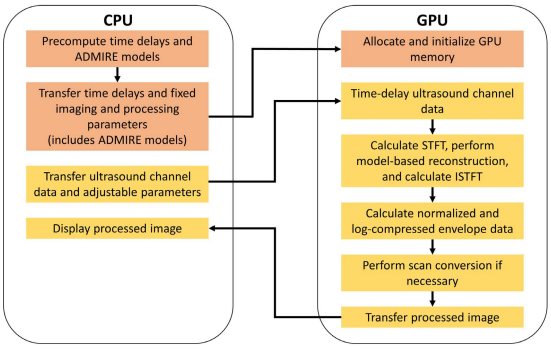

Fig. 2. Diagram of the processing pipeline for the GPU implementation of ADMIRE. Peach colored boxes represent stages that are only performed once. Yellow colored boxes represent stages that are performed for every image frame.

sections further describe the stages of this pipeline in greater detail.

C. Data Transfer and Time-Delaying Channel Data

In this pipeline, the undelayed ultrasound channel data and various imaging and processing parameters are transferred to the GPU. Once on the GPU, the channel data is transferred to a cudaArray that is then bound to texture memory. This type of GPU memory is used to time-delay the channel data because fast linear interpolation can be performed in hardware. The delays that are used for this process are precomputed on a CPU and then transferred to the GPU. To apply the delays, a CUDA kernel is used. In terms of the grid and block structure of the kernel, a 2-D grid structure and a 1-D block structure are utilized, where each block in the grid corresponds to an image pixel position and each thread within a single block obtains the time-delayed channel data value for a single aperture element. Note that if it is desired, the delays that are used in the delay kernel can also be directly computed on the GPU using a separate CUDA kernel.

D. Short-Time Fourier Transform

After the channel data is delayed, the STFT is calculated using a 0% window overlap. A window overlap of 0% is used due to the fact that using a smaller window overlap requires less GPU VRAM memory to store the STFT data, and it improves the computational efficiency of ADMIRE. Moreover, it has previously been shown that performing ADMIRE with a reduced window overlap does not result in noticeable qualitative image degradation [22]. To calculate the STFT, a kernel is utilized to first perform zero-padding of the data in order to be able to interpolate additional frequency bins within each window. A 3-D grid structure and a 2-D block structure are utilized by this kernel. The windows for each beam are handled in subgroups. Each block handles zero-padding for the same column in every subgroup window for one beam, where each column corresponds to a specific aperture element. Therefore, the total number of blocks is equal to the number of

columns per window times the number of window subgroups per beam times the number of beams. Within a single block, the number of threads is equal to the number of windows per subgroup times the zero-padded STFT length. Each thread handles obtaining the data value for one entry of one window column. The threads that do not store zeros corresponding to zero-padding also multiply their obtained values by windowing coefficients that are transferred to the GPU. In this case, a rectangular window that keeps all of the samples is used. Note that the last window subgroup for each beam may contain fewer windows than the other window subgroups. Therefore, some CUDA warps corresponding to certain blocks can have a number of threads that are not actually performing computational operations. For similar reasons, some of the other CUDA kernels in the processing pipeline can also have blocks where some of their corresponding CUDA warps have a number of idle threads.

Once the data for each STFT window has been arranged, the NVIDIA cufft library is utilized to perform 1-D Fourier transforms along the columns of the windows. After doing this, each window contains the aperture domain data for multiple frequencies, and the completion of this step corresponds to the completion of Fig. 1(A). In ADMIRE, a subset of the frequencies corresponding to the pulse bandwidth are typically fit. Therefore, a kernel is utilized to obtain the data for only the frequencies that are selected for model-based reconstruction. For example, if the frequencies that are to be fit correspond to rows 3–5 in each STFT window, then this kernel will obtain those rows of aperture domain frequency data for every window in each beam. A 3-D grid structure is utilized along with a 1-D block structure, where each block handles obtaining one row of data in one STFT window for a single beam and each thread obtains the real and imaginary components for its corresponding position in the row. Therefore, the total number of blocks is equal to the number of selected frequencies per STFT window times the number of STFT windows per beam times the number of beams, and the number of threads per block is equal to the number of columns per STFT window. In addition, when these rows of data are obtained and stored, their real and imaginary components are separated such that all of the real components for a row are stored first followed by all of the imaginary components. This separation of the real and imaginary components is specifically done for the model fitting and reconstruction stage that follows.

E. Model Fitting and Reconstruction

To prepare for model fitting and reconstruction, another kernel is utilized. One purpose of this kernel is to account for aperture growth if it is applied. In the previous step, rows of aperture domain frequency data were selected from the STFT windows. Therefore, in this step, binary masks that are precomputed on the CPU and transferred to the GPU are used, where each mask determines which aperture element samples to remove for a given row of aperture domain frequency data. For example, suppose three rows of aperture domain frequency data are selected from a given STFT window. When working with channel data, the depth that a set of aperture samples corresponds to is used in the aperture growth computation in order to determine how many aperture elements to remove to obtain a given F-number. In this case, the center depth that the STFT window corresponds to is used in the aperture growth computation. This means that each row of aperture domain frequency data from the same STFT window will have an identical binary mask. After applying the masks, the result is that the length of the rows of aperture domain frequency data from different STFT windows will vary depending on how many samples have been removed.

Following this, the same kernel is used to divide each row of aperture domain frequency data by its standard deviation [$(1/N)$ formula], where N is two times the number of aperture elements that are left after aperture growth is applied (factor of two accounts for the real and imaginary components being separated). In addition, the λ parameter that is used when performing linear regression with elastic-net regularization is calculated for each row of aperture data. This is calculated as $c_\lambda((\boldsymbol{y}^T\boldsymbol{y})/N)^{1/2}$, where c_λ is a scaling factor that is typically set to 0.0189 and $\boldsymbol{y}$ is one set of aperture data before being standardized. To account for the standardization, each λ value is also divided by its respective aperture data set standard deviation. Once these steps are complete, Fig. 1(B) can be performed. In terms of the structure of the kernel, a 1-D grid structure and a 1-D block structure are utilized, where each block contains 32 threads. Each thread performs the aforementioned operations for one row of aperture domain frequency data, and as many blocks are used as are required to perform the operations for all of the selected rows of aperture domain frequency data across all of the STFT windows and beams. Note that subsequent references to $\boldsymbol{y}$ refer to one set of aperture domain frequency data after standardization.

Once all of the rows of aperture domain frequency data have been prepared, each row is fit with its corresponding model matrix. The purpose of this is to determine how the different model predictors contribute to a given set of aperture domain frequency data so that the contributions of the predictors corresponding to multipath and off-axis scattering can be removed. As previously stated, linear regression with elastic-net regularization is utilized to fit each model matrix to its corresponding set of aperture domain frequency data. In the CPU implementation of ADMIRE, the model fits are performed 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 GPU implementation of cyclic coordinate descent [25], which is the optimization method that is used to estimate the model coefficients for each fit by minimizing the objective function shown in (1). In this case, P is the number of predictors in the model matrix. Note that we have also developed a C MEX-file to perform cyclic coordinate descent for the CPU implementation, but both of the custom implementations of cyclic coordinate descent were inspired by the glmnet software package [26]–[28], which is the package that was originally used to perform the model fits for the CPU implementation

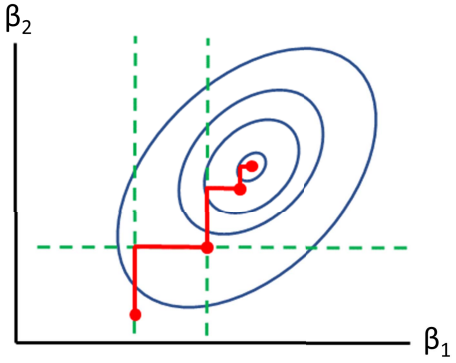

Fig. 3. Example of the cyclic coordinate descent optimization method. Each red point represents the objective function value at the end of a single iteration. The solid red lines represent minimizing the objective function with respect to one model coefficient at a time.

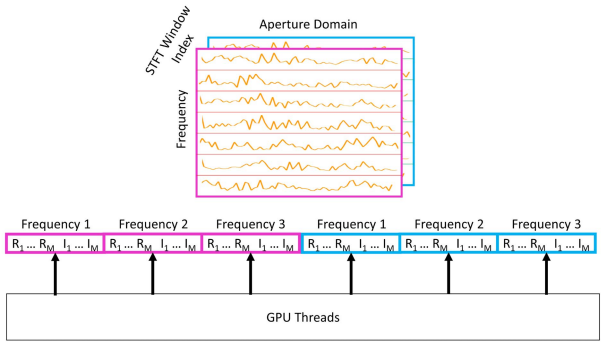

Fig. 4. Example of how each GPU thread performs one model fit using one set of aperture domain frequency data. In this case, three frequencies are being reconstructed with model fits in each STFT window. For each set of aperture domain frequency data, the real components are stored first followed by the imaginary components. *M* is equal to the number of aperture elements that are used for the STFT window after taking aperture growth into account. Due to this, the value of *M* can vary across STFT windows. The color of each aperture data block represents the STFT window that it corresponds to.

of ADMIRE.

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \frac{1}{2N} \sum_{i=1}^{N} \left(\boldsymbol{y}_i - \sum_{j=1}^{P} \boldsymbol{X}_{ij} \boldsymbol{\beta}_j \right)^2 + \lambda \left(\alpha \|\boldsymbol{\beta}\|_1 + \frac{(1-\alpha)\|\boldsymbol{\beta}\|_2^2}{2} \right) \quad (1)$$

A single iteration of cyclic coordinate descent requires minimizing the objective function with respect to one model coefficient at a time, and the update for a single predictor is shown in (2), where S is a soft-thresholding function. All of the coefficients are cycled through, and this is done for multiple iterations until specified convergence criteria are met. This process is illustrated in Fig. 3. Note that $\hat{\boldsymbol{y}}_i^{(j)}$ represents the value of $\hat{\boldsymbol{y}}_i$ leaving predictor j out. The derivation is provided in the Appendix.

$$\hat{\boldsymbol{\beta}}_j \leftarrow \frac{S\left(\frac{1}{N}\sum_{i=1}^{N} \boldsymbol{X}_{ij}\left(\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i^{(j)}\right), \lambda\alpha\right)}{\frac{1}{N} + \lambda(1-\alpha)}$$

$$S(z,\gamma) = \begin{cases} z-\gamma, & \text{if } z > 0 \text{ and } \gamma < |z| \\ z+\gamma, & \text{if } z < 0 \text{ and } \gamma < |z| \\ 0, & \text{if } \gamma \geq |z| \end{cases} \quad (2)$$

Cyclic coordinate descent is executed on the GPU by having each thread perform a fit involving a specific set of aperture domain frequency data. The kernel that performs the model fits utilizes a 1-D grid structure and a 1-D block structure, where each block contains 32 threads. As many blocks are initialized as are required for handling all of the sets of aperture domain frequency data across all of the STFT windows and beams. For example, suppose there are 128 beams, with each beam containing 300 STFT windows and each STFT window requiring the aperture data for three frequencies to be reconstructed with model fits. In this case, 115 200 total model fits must be performed, so each one of these model fits will be assigned to one thread as shown in Fig. 4. All of the model matrices can be precomputed and transferred to the GPU only once because the same models can be reused for different image frames. The predictors in each model matrix are also normalized such that $\sum_{i=1}^{N} \boldsymbol{X}_{ij}^2 = 1$ when they are precomputed.

Each thread performs cyclic coordinate descent until one of two convergence criteria is first met. The first convergence criterion is a limit to the maximum number of iterations of cyclic coordinate descent. The second convergence criterion is a tolerance for the maximum weighted sum of squares of the changes in the fitted values between iterations of cyclic coordinate descent. Essentially, each time the model coefficient of a predictor is updated, the weighted sum of squares of the changes in the fitted values due to the update is calculated as $(1/N)\sum_{i=1}^{N}(\boldsymbol{X}_{ij}\hat{\boldsymbol{\beta}}_j^{(k)} - \boldsymbol{X}_{ij}\hat{\boldsymbol{\beta}}_j^{(k+1)})^2$, where $\hat{\boldsymbol{\beta}}_j^{(k)}$ is the value of the model coefficient before the update and $\hat{\boldsymbol{\beta}}_j^{(k+1)}$ is the value of the model coefficient after the update. This term is calculated for all of the model coefficient updates, and if the maximum across all of the updates is less than the specified tolerance, the execution of cyclic coordinate descent is stopped. Note that the observation weights are all 1 in this case. Moreover, due to the fact that the predictors are normalized, the tolerance criterion can be simplified to $(1/N)(\hat{\boldsymbol{\beta}}_j^{(k)} - \hat{\boldsymbol{\beta}}_j^{(k+1)})^2$. Now, once one of the convergence criteria is met and the optimization algorithm is stopped, each thread unstandardizes the model coefficients for its corresponding fit and then reconstructs its set of decluttered aperture domain frequency data by only using the predictors corresponding to a specific ROI as shown in (3). By doing so, the contributions of the predictors that correspond to off-axis scattering and multipath scattering are eliminated.

$$\boldsymbol{y}_{\text{ROI}} = \boldsymbol{X}_{\text{ROI}} \hat{\boldsymbol{\beta}}_{\text{ROI}} \quad (3)$$

F. GPU Implementation Optimizations

In order to increase the speed of each instance of cyclic coordinate descent in our GPU implementation, a number of optimizations are utilized. One optimization is that due to the fact that each model matrix represents an aperture domain model, one model matrix can be used for multiple model fits. For example, the first selected row of aperture

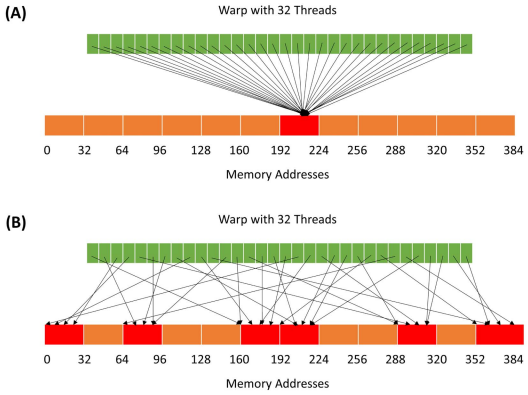

Fig. 5. (A) Example of an L1 cache memory access pattern where all of the threads within one warp require data that is located in one sector of memory, which means that only one memory transaction is required. In particular, the threads require the same memory address within the memory sector, which results in a broadcast operation. (B) Example of an L1 cache memory access pattern where the threads within one warp require data that is located in memory addresses that are not contiguous, which means that multiple memory transactions are required. Each cache line consists of four memory sectors that are each 32 bytes for a total of 128 bytes.

domain frequency data for STFT window 1 for beam 1 uses the same model matrix as the first selected row of aperture domain frequency data for STFT window 1 for all of the other beams. This can be used for optimization because, as previously stated, the kernel that performs the model fits assigns one model fit to each thread, where each block contains 32 threads. Due to this, the kernel is set up such that each thread within the same block performs a model fit for the same row number and STFT number, but the only factor that varies between the threads is the beam number. For example, the first thread block will perform the model fits for the first selected row of data for STFT window 1 for beams 1–32, and the second block will perform the model fits for the first selected row of data for STFT window 1 for beams 33–64. Depending on the number of fits, hundreds to thousands of these blocks may be required. However, by having the threads within a given block use the same model matrix instead of different model matrices, increased coalesced memory access is achieved. In other words, the number of memory transactions required to service the threads is reduced, which means that GPU memory bandwidth is more efficiently utilized and latency due to memory transactions is reduced. This memory access pattern is illustrated in Fig. 5(A). A less efficient memory access pattern that might occur if the threads within a block accessed different model matrices is also illustrated in Fig. 5(B).

In addition to this optimization, another optimization is that an efficient computation scheme is utilized to calculate the residual, which in this case, does not include the contribution of the current predictor for which the model coefficient is being updated. One way of calculating this is by first obtaining $\hat{\boldsymbol{y}}^{(j)}$ by doing the matrix multiplication of the model matrix and the vector of model coefficients, where both exclude the current predictor. This can then be subtracted from $\boldsymbol{y}$ to obtain the residual without the current predictor. This method is inefficient due to the fact that only one model coefficient corresponding to the current predictor is being updated at a time in cyclic coordinate descent, so a more efficient method is to only take into account how the update to this single model coefficient impacts the residual. For example, for a given model fit, we initialize all of the model coefficients to 0. Therefore, at the beginning of the first iteration of cyclic coordinate descent, the residual is given by $\boldsymbol{y}$. All of the predictors are then cycled through. For each predictor, the impact of its model coefficient is removed from the residual in order to first calculate the residual without the predictor. Following this, the correlation between the predictor's observations and the residual without the predictor is computed. The coordinate descent update of the predictor's model coefficient is then performed, and the impact of the updated model coefficient is incorporated into the residual. This process continues until one of the convergence criteria is met. By using this method for calculating the residual, performance is improved due to the fact that the number of computations is reduced, and the number of memory accesses is also reduced. Pseudocode for the algorithm is provided below. The C MEX-file that is used to perform cyclic coordinate descent for the CPU implementation of ADMIRE also uses this optimization. For the GPU implementation, this scheme is further improved by storing the residual vector for each model fit in shared memory, which has lower memory access latency than global memory. Shared memory is also used to store the reconstructed aperture domain frequency data for each model fit when performing (3). In addition, for the kernel that prepares the data for model fitting and reconstruction, shared memory is used to store the sets of aperture domain frequency data during the process of standardizing them.

G. Inverse Short-Time Fourier Transform

After all of the selected rows of aperture domain frequency data have undergone model-based reconstruction, the ISTFT of the data is calculated. However, prior to this step, one additional CUDA kernel is utilized. As previously stated, model fits are only performed for the frequencies that correspond to the pulse bandwidth. To reduce computational time, the aperture domain frequency data corresponding to negative frequencies is not fit. Therefore, this kernel stores the complex conjugate of the reconstructed aperture domain frequency data for the selected positive frequencies into the rows that correspond to their negative frequencies. A 3-D grid structure and a 1-D block structure are utilized for the kernel. The number of blocks is equal to the number of frequencies per STFT window that are reconstructed times the number of STFT windows per beam times the number of beams, and the number of threads within a single block is equal to the number of columns per STFT window. Each thread stores the data for one position in a specific row of reconstructed data, and it also stores the complex conjugate in the same position for the corresponding negative frequency row, if there is one. If a thread's particular position in the row is eliminated due to aperture growth in the kernel that prepares the data for model fitting and reconstruction, then the thread does not store any

Algorithm 1 Cyclic Coordinate Descent

Initialize $\hat{\boldsymbol{\beta}}_j \leftarrow 0$ **for** $j \leftarrow 1$ to P
$\boldsymbol{r} \leftarrow \boldsymbol{y}$, where $\boldsymbol{r}$ is the residual vector
while convergence criteria not met **do**
  for $j \leftarrow 1$ to P **do**
    $\rho_j \leftarrow 0$
    if $\hat{\boldsymbol{\beta}}_j \neq 0$ **then**
      for $i \leftarrow 1$ to N **do**
        $\boldsymbol{r}_i \leftarrow \boldsymbol{r}_i + \boldsymbol{X}_{ij}\hat{\boldsymbol{\beta}}_j$
        $\rho_j \leftarrow \rho_j + \boldsymbol{X}_{ij}\boldsymbol{r}_i$
      end for
    else
      $\rho_j \leftarrow \rho_j + \boldsymbol{X}_{ij}\boldsymbol{r}_i$
    end if
    $\hat{\boldsymbol{\beta}}_j \leftarrow \frac{S(\frac{1}{N}\rho_j, \lambda\alpha)}{\frac{1}{N}+\lambda(1-\alpha)}$
    if $\hat{\boldsymbol{\beta}}_j \neq 0$ **then**
      for $i \leftarrow 1$ to N **do**
        $\boldsymbol{r}_i \leftarrow \boldsymbol{r}_i - \boldsymbol{X}_{ij}\hat{\boldsymbol{\beta}}_j$
      end for
    end if
  end for
end while

data. After the data is stored, Fig. 1(C) can be performed. Note that the aperture domain frequency data for all other frequencies besides the aforementioned ones is zeroed out.

The cufft library is used to perform 1-D inverse Fourier transforms along the columns of the STFT windows for all of the beams. As previously stated, the window overlap used for the STFT is 0%, and a rectangular window that keeps all of the window samples is used. Therefore, performing just the 1-D inverse Fourier transforms gives the ISTFT of the data in this case. In addition, due to zero-padding being used for the STFT, a kernel is used to remove the positions corresponding to zero-padding after performing the ISTFT. The kernel utilizes a 3-D grid structure and a 1-D block structure. Each block handles obtaining a row of data for one window of decluttered channel data for a given beam, and each thread within a block stores the data for its corresponding position in the row. Therefore, the number of blocks is equal to the number of rows per window that do not correspond to zero-padding times the number of windows per beam times the number of beams, and the number of threads is equal to the number of columns per window.

H. Image Calculation Using Decluttered Channel Data

To obtain the summed RF data from the channel data, a summing kernel is used. This kernel utilizes a 1-D grid structure and a 1-D block structure. The number of blocks is equal to the number of depth samples, and the number of threads within a block is equal to the number of beams. Each thread sums the aperture data for its corresponding image pixel position. Whenever possible, a more optimized summing kernel that uses principles from an existing parallel reduction algorithm [29] is utilized. In this implementation, a 2-D grid structure and a 1-D block structure are used. Each block corresponds to an image pixel position, and the number of threads within a block is equal to the number of aperture elements. The aperture element samples are stored in shared memory in this version of the kernel.

Once the summed RF data is obtained, the envelope data is calculated by first taking the Hilbert transform. The process for taking the Hilbert transform follows the discrete algorithm [30], which is also used by MATLAB. On the GPU, this involves computing the 1-D Fourier transform of each beam. A kernel is then used to weight the frequency bins. A 1-D grid structure and a 1-D block structure are utilized. The number of blocks is equal to the number of frequency bins for one beam, and the number of threads within a block is equal to the number of beams. Each thread weights the frequency bin for its corresponding beam. Once the data is weighted, the 1-D inverse Fourier transform of each beam is computed to obtain IQ data. The magnitude of the IQ data is computed using a separate kernel that utilizes a 1-D grid structure and a 1-D block structure, where the number of blocks is equal to the number of depth samples and the number of threads is equal to the number of beams. Each thread computes the magnitude of the IQ data sample for its corresponding image pixel position.

In order to perform envelope normalization and log compression, a kernel is first used to obtain the maximum value of the envelope data. This kernel uses one block of threads, where the number of threads is equal to the number of beams. Each thread finds the maximum envelope data value for its corresponding beam and stores it into shared memory. Once this is complete, one thread is utilized to find the global maximum across all of the beam maximum values. This maximum value is utilized in a second kernel to normalize the envelope data. Log compression is applied in this second kernel as well. The kernel utilizes a 1-D grid structure and a 1-D block structure, where the number of blocks is equal to the number of depth samples and the number of threads within each block is equal to the number of beams. Each thread normalizes and log-compresses the data value for its corresponding image pixel position. Once normalization and log compression are performed, the processed image is transferred back to the host in order to be displayed. In cases such as when using a curvilinear probe, scan conversion of the image data is also performed on the GPU by using a series of GPU kernels before transferring the processed image. Note that in Fig. 2, image display is listed under CPU due to the fact that MATLAB is used to display the processed image. However, MATLAB ends up sending the image back to the GPU in order to actually display it.

I. CPU ADMIRE Versus GPU ADMIRE Benchmarks

To compare the GPU and CPU implementations of ADMIRE, benchmarks were performed utilizing a Field II [31], [32] cyst simulation data set with added white Gaussian noise. The power of the noise was scaled to be 0 dB relative to the power of the channel data. All of the parameters required for ADMIRE, such as the model matrices and the channel data time delays, were precomputed. ICA was applied to

TABLE I
IMAGING/PROCESSING PARAMETERS USED FOR BENCHMARK

Imaging/Processing Parameter	Simulated Data Set
Depth Samples	2,400
Elements per Beam	128
Beams	128
f0 (MHz)	5.8125
fs (MHz)	23.25
Padded STFT Window Length	12
STFT Window Overlap	0%
Frequencies Fit per STFT Window	3
α for Regularization	0.9
c_λ for Regularization	0.0189
Coordinate Descent Max Iterations	100,000
Coordinate Descent Tolerance	0.1
Aperture Growth	Not Applied

reduce the sizes of the model matrices. For running ADMIRE on a single GPU, an NVIDIA GeForce GTX 1080 Ti GPU was used, and an NVIDIA GeForce RTX 2080 Ti GPU was also used to see how the processing time changes depending on the GPU. Multi-GPU processing using both GPUs was also performed. For running ADMIRE on multiple GPUs, the computations were split across the GeForce GTX 1080 Ti GPU and the GeForce RTX 2080 Ti GPU, with 32 beams distributed to the 1080 Ti GPU and 96 beams distributed to the 2080 Ti GPU due to the fact that it is more powerful. To estimate the best distribution of beams, processing was performed using different distributions, and the one that resulted in the fastest processing time was utilized to perform the benchmark. When the beams are distributed across the two GPUs, the sets of channel data are time-delayed and reconstructed with ADMIRE separately. The envelope data of each set is calculated separately as well. However, the two sets of envelope data are recombined onto one GPU before the envelope data is normalized and log-compressed. The host computer used for the benchmarks contained dual Intel (Intel Corporation, Santa Clara, CA) Xeon Silver 4114 CPUs @ 2.20 GHz with 10 cores each. The MATLAB programming language was used for the CPU implementation of ADMIRE, and to perform cyclic coordinate descent for this implementation, a MEX-file written in C was utilized. The number of computational threads within MATLAB was also set to 1 in order to perform serial processing.

The imaging and processing parameters can be seen in Table I. Note that Table I shows the number of depth samples as 2400, but ADMIRE was applied from depth sample 13 to depth sample 2400. This means that the first 12 depth samples had only DAS applied to them by default. In regard to numerical precision, for the CPU implementation, the calculations performed in MATLAB and the calculations performed in the C MEX-file both used double precision. For the GPU implementation, the calculations were performed using single precision because the 1080 Ti and 2080 Ti GPUs have fewer FP64 units than FP32 units, which means that using double precision results in reduced computational speed. Moreover, using double precision requires more memory resources, which also reduces performance. The processing time was measured using MATLAB's built-in timing capabilities and averaged across 10 runs for the CPU case and 100 runs for the GPU cases due to the processing times being smaller. One warmup run was performed before performing the runs for each benchmark. In addition, contrast ratio values were calculated using (4), and the relative error between the sets of normalized envelope data before log compression for the CPU and GPU implementations was calculated using (5).

$$\mathrm{CR} = -20\log_{10}\left(\frac{\mu_{\mathrm{lesion}}}{\mu_{\mathrm{background}}}\right) \tag{4}$$

$$\text{Relative Error} = 20\log_{10}\left(\frac{\|\mathrm{Env}_{\mathrm{CPU}} - \mathrm{Env}_{\mathrm{GPU}}\|_2}{\|\mathrm{Env}_{\mathrm{CPU}}\|_2}\right) \tag{5}$$

J. Effect of Imaging and Processing Parameters on Run Time

To assess the effect of imaging and processing parameters on the processing time for the GPU implementation of ADMIRE, the same cyst simulation data set from the previous section was utilized. The example parameters that were varied were the number of depth samples, the number of elements per beam (this refers to the transducer elements whose signals are summed together in order to form one A-line), the number of beams, the tolerance for cyclic coordinate descent, the c_λ value that is used to calculate the value of λ for elastic-net regularization, and the α value that is used in elastic-net regularization. One parameter was varied at a time while holding the other parameters constant at their values listed in Table I. For each parameter value case, the GPU processing time in seconds was measured and averaged across 100 runs. One warmup run was performed before performing the runs for each benchmark. The GPU benchmarks were performed using an NVIDIA GeForce GTX 1080 Ti GPU only, using an NVIDIA GeForce RTX 2080 Ti GPU only, and using multi-GPU processing with both of the GPUs. The same host computer from the previous section was utilized. For multi-GPU processing, the distribution of beams across the two GPUs varied depending on the case. The best distribution was determined for each case using the method that was previously described, and each benchmark was then performed using the selected distribution. For all of these cases, ADMIRE was not applied to the first 12 depth samples, so DAS was applied to them by default.

In addition, for the cases involving varying the number of depth samples, the number of elements per beam, and the number of beams, NVIDIA's Nsight Systems profiling tool was used to obtain example processing times for individual stages of the GPU processing pipeline. This was not done for the cases involving varying the tolerance for cyclic coordinate descent, the c_λ value, and the α value because these parameters only impact the CUDA kernel that performs the model fits. For each parameter value case, the profiler times for one run using the 2080 Ti GPU were obtained. Note that a given stage may consist of more than one CUDA kernel as described in the corresponding sections for the stages.

K. Verasonics Imaging

To perform imaging with the GPU implementation of ADMIRE, a Verasonics Vantage 128 ultrasound research system (Verasonics, Kirkland, WA) was utilized. The host computer that was used contained dual Intel Xeon E5-2650 v2 CPUs @ 2.60 GHz with 8 cores each. This system also contained one NVIDIA GeForce RTX 2080 Ti GPU. Verasonics sequences were developed in order to collect ultrasound channel data and process it using the GPU implementation of ADMIRE. An additional CUDA kernel is called as part of the GPU processing pipeline in this case in order to reshape the Verasonics channel data buffer that is transferred to the GPU. The data type of the samples in this buffer are also converted from int16 to float within the kernel that performs reshaping. The data is then processed as previously described. Once processed, the normalized and log-compressed envelope data (also scan-converted if required) is returned to the host system in order to be displayed. Note that in the case of the Field II data set that was discussed in the previous sections, the data was already in the correct shape, and it was converted from double precision to single precision before it was transferred to the GPU. Therefore, the kernel that performs reshaping and data type conversion was not required.

The first sequence that was written was for an L7-4 linear transducer array, and it consisted of performing a walked aperture scan with focused transmits. The second sequence that was written was for a C5-2 curvilinear transducer array, and it also consisted of performing a walked aperture scan with focused transmits. For both of the sequences, the Verasonics hardware and software sequencers were synchronized so that one frame of channel data was collected and processed before the next frame was acquired. The GPU implementation of ADMIRE was called as an external processing event in the Verasonics sequence pipeline. Custom GUI controls were also added to the Verasonics GUI in order to allow for certain ADMIRE parameters to be changed during imaging. These parameters were the tolerance for cyclic coordinate descent, the maximum number of iterations of cyclic coordinate descent to perform, the c_λ value, and the α value. All of the other parameters that are required for ADMIRE, such as the model matrices and channel data time delays, were precomputed. They were then loaded onto the GPU at the beginning of each sequence. ICA was applied to reduce the sizes of the model matrices.

In regard to scanning, the L7-4 sequence was used to scan the carotid artery of a human subject, and the C5-2 sequence was used to scan abdominal organs, such as the liver and kidneys, of a human subject. The scans of the human subject were performed with approval from the Vanderbilt University Institutional Review Board. The imaging and processing parameters for each sequence are shown in Table II. The parameters denoted with (Def.) are the adjustable ADMIRE parameters, and the listed values are the default values for when the sequences begin. Note that for the L7-4 sequence, ADMIRE was applied from depth sample 25 to depth sample 2664, which means that DAS was applied to the first 24 and last three depth samples by default. In addition, for the C5-2 sequence, ADMIRE was applied from depth sample 7 to depth sample 2154, which means that DAS was applied to the first six and last three depth samples by default. To determine the frame rate, the time required to acquire, process, and display 100 image frames was measured within MATLAB.

TABLE II
IMAGING/PROCESSING PARAMETERS FOR EACH VERASONICS SEQUENCE

Imaging/Processing Parameter	L7-4 Sequence	C5-2 Sequence
Depth Samples	2,667	2,157
Elements per Beam	65	65
Beams	64	64
f0 (MHz)	5.2083	3.125
fs (MHz)	20.8333	12.5
Padded STFT Window Length	12	12
STFT Window Overlap	0%	0%
Frequencies Fit per STFT Window	3	3
α for Regularization (Def.)	0.9	0.9
c_λ for Regularization (Def.)	0.0189	0.0189
Coordinate Descent Max Iterations (Def.)	100,000	100,000
Coordinate Descent Tolerance (Def.)	0.1	0.1
Aperture Growth	Not Applied	Not Applied

L. Open-Source Code Repository

In order to allow for use of ADMIRE in the community, an open-source code repository has been created at the following link: https://github.com/VU-BEAM-Lab/ADMIRE. This code repository includes a CPU implementation of ADMIRE and a GPU implementation for use with one GPU. Example Verasonics scripts to perform imaging are also included. These scripts are for the sequences that are described above. In addition, the repository includes tutorials that demonstrate how to use the code.

III. RESULTS

A. CPU ADMIRE Versus GPU ADMIRE Benchmarks

Fig. 6 shows the simulated cyst data set processed with DAS using different GPU cases, ADMIRE using different GPU cases, and ADMIRE using a CPU. For DAS, the beam distribution that was used for the multi-GPU case was that 18 beams were distributed to the 1080 Ti GPU and 110 beams were distributed to the 2080 Ti GPU. The average processing times and standard deviations across runs are displayed on each image, and the contrast ratio values are also shown. The processing times for DAS include time-delaying the channel data and computing the normalized and log-compressed envelope data. The processing times for the GPU and CPU implementations of ADMIRE include time-delaying the channel data, reconstructing the channel data using ADMIRE, and computing the normalized and log-compressed envelope data. The transfer time for transferring the channel data from host memory to GPU memory along with the transfer time for transferring the normalized and log-compressed envelope data from GPU memory to host memory were included in the DAS and ADMIRE GPU processing times. The inclusion of these transfer times is the main reason the processing times are in the range of 37–42 ms for the DAS GPU cases. For example, using NVIDIA's Nsight Systems profiling tool to analyze the times for individual steps in the processing pipeline, it was

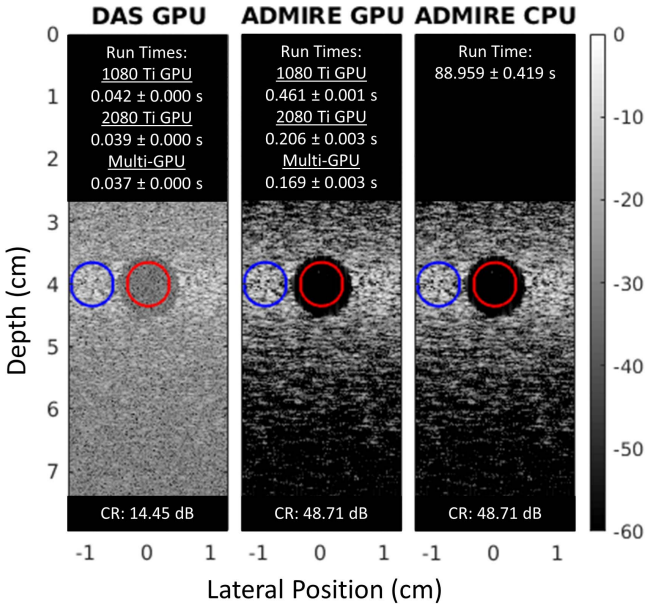

Fig. 6. Field II cyst simulation data set with added white Gaussian noise processed with DAS using different GPU cases (left), ADMIRE using different GPU cases (center), and ADMIRE using a CPU (right). The power of the noise was scaled to be 0 dB relative to the power of the channel data. The average run times in seconds ($N = 10$ for CPU and $N = 100$ for GPU) are displayed at the top of the images along with the standard deviations, and the contrast ratio values in dB are displayed at the bottom. The cyst and background masks that were used for calculating the contrast ratio values are displayed as well.

determined that the actual time spent performing compute operations for the DAS cases was within the approximate range of 3–6 ms. This means that the time taken for memory operations was the primary performance bottleneck. Note that memory operations also include transferring the channel data to a cudaArray, and in the case of ADMIRE, calls to the cudaMemset command. However, they do not include memory transactions that take place within CUDA kernels. For the GPU implementation of ADMIRE, the time for transferring most of the ADMIRE parameters was not included in the processing times because in a real-time imaging setting, these parameters only need to be transferred once at the beginning. After this, they can be reused for all of the imaging frames during scanning. The only parameters whose transfer times were included were the tolerance for cyclic coordinate descent, the maximum number of iterations of cyclic coordinate descent to perform, the c_λ value, and the α value. This is because these parameters are the ones that are most likely to be changed during real-time imaging.

Moreover, for this case, the relative error between the sets of normalized envelope data before log compression for the GPU and CPU implementations of ADMIRE is −63.68 dB. This error is most likely due to a combination of factors such as architectural differences between the GPU and CPU potentially causing differences in the results of numerical calculations, using single precision for the GPU calculations while using double precision for the CPU calculations, the fact that parallelizing certain algorithms can result in different amounts of round-off error when compared to their serial implementations, and the fact that the interpolation coefficients are stored in 9-bit fixed point format when performing interpolation with texture memory on the GPU. However, this is acceptable because there are no noticeable qualitative differences between the image produced by the GPU implementation and the image produced by the CPU implementation.

Nevertheless, as a test, the channel data for the data set that was used to create Fig. 6 was time-delayed on the GPU. This delayed data was then used in the CPU implementation of ADMIRE in order to take into account texture memory interpolation on the GPU being coarser. The calculations for the CPU implementation were also performed using single precision instead of double precision in order to take into account numerical precision. When these two factors are taken into account, the relative error is reduced to −129.16 dB. Note that for calculating the relative error values, all of the sets of normalized envelope data were converted to double-precision if they were not already in this precision in order to ensure that the same numerical precision was used for the calculation in (5).

B. Effect of Imaging and Processing Parameters on Run Time

In regard to the effect of imaging and processing parameters on the GPU implementation of ADMIRE, Fig. 7 demonstrates that there is a linear relationship between processing time and three of the parameters that were studied (number of depth samples, number of elements per beam, and number of beams). For example, reducing the number of depth samples by a factor of 2 results in the processing time also being reduced by a factor of 2, reducing the number of elements per beam by a factor of 2 reduces the processing time by a factor of 3.5–4.5, and reducing the number of beams by a factor of 2 reduces the processing time by a factor 1.5–2 (factors were rounded to the nearest 0.5 increments). Now, in contrast to the first three parameters, the processing time does not appear to have a linear relationship with the tolerance for cyclic coordinate descent, the c_λ value, and the α value. Fig. 7 shows that the processing time decreases as each of these parameters are increased until it eventually levels out.

Understanding the effects of these parameters is important, and it allows us to estimate performance. For example, as an additional test, an extra benchmark case was performed using the multi-GPU implementation with the same data set. The number of depth samples was set to 1200, the number of elements per beam was set to 64, the number of beams was set to 64, the tolerance for cyclic coordinate descent was set to 0.1, the value for c_λ was set to 0.0189, and the α value was set to 0.9. The determined relationships between the parameters and the processing time were used to estimate the processing time for this case. The original data had 2400 depth samples, 128 elements per beam, 128 beams, a tolerance of 0.1, a c_λ value of 0.0189, and an α value of 0.9. The reductions in the number of depth samples, the number of elements per beam, and the number of beams correspond to processing time reduction factors of 2, 3.5–4.5, and 1.5–2, respectively. Multiplying these factors together, the processing

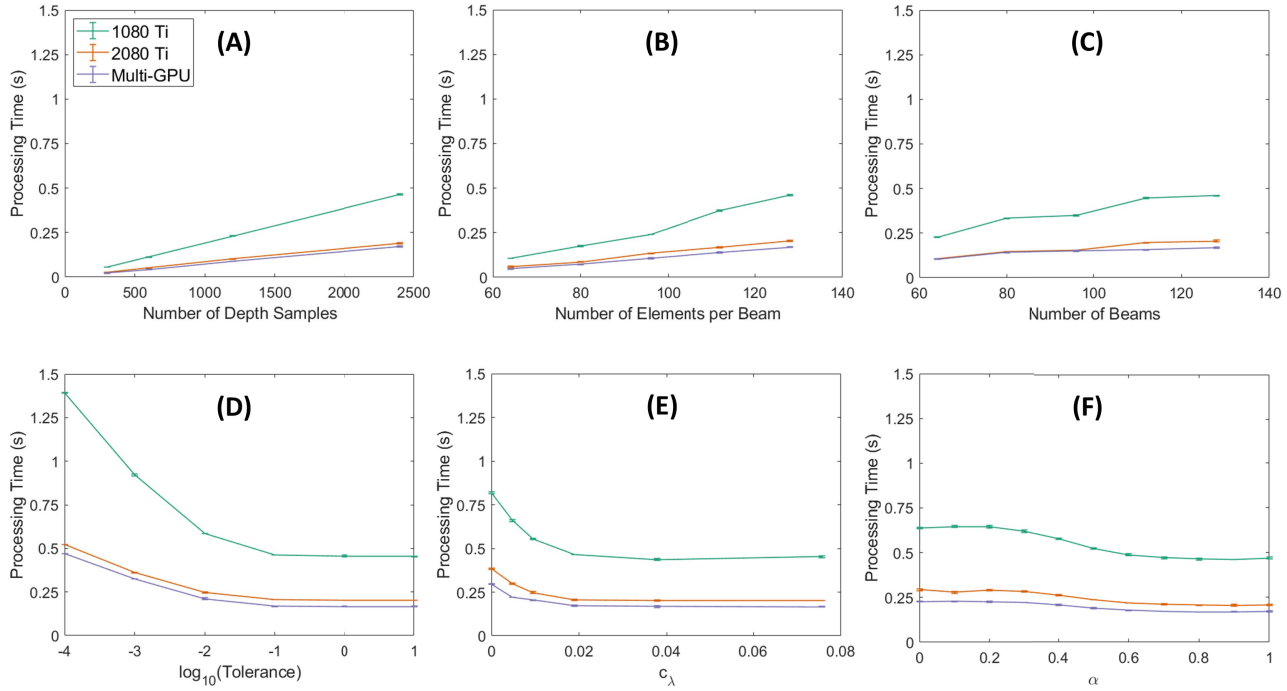

Fig. 7. GPU processing time as a function of **(A)** the number of depth samples, **(B)** the number of elements per beam, **(C)** the number of beams, **(D)** the tolerance value for cyclic coordinate descent (displayed using a logarithmic scale), **(E)** the c_λ value that is used to calculate λ for elastic-net regularization, and **(F)** the α value that is used in elastic-net regularization. Error bars show the standard deviations ($N = 100$). When varying one parameter, the other parameters were held constant at their default values listed in Table I. Note that for the case where $\alpha = 0$ in **(F)**, cyclic coordinate descent was still used to perform the model fits even though an analytic solution does exist when performing linear regression with L2-regularization only. In addition, for the multi-GPU case where the number of beams is equal to 64 in **(C)**, the beam distribution that resulted in the fastest time was distributing all of the beams to the 2080 Ti GPU. The color scheme for this figure was selected using [33].

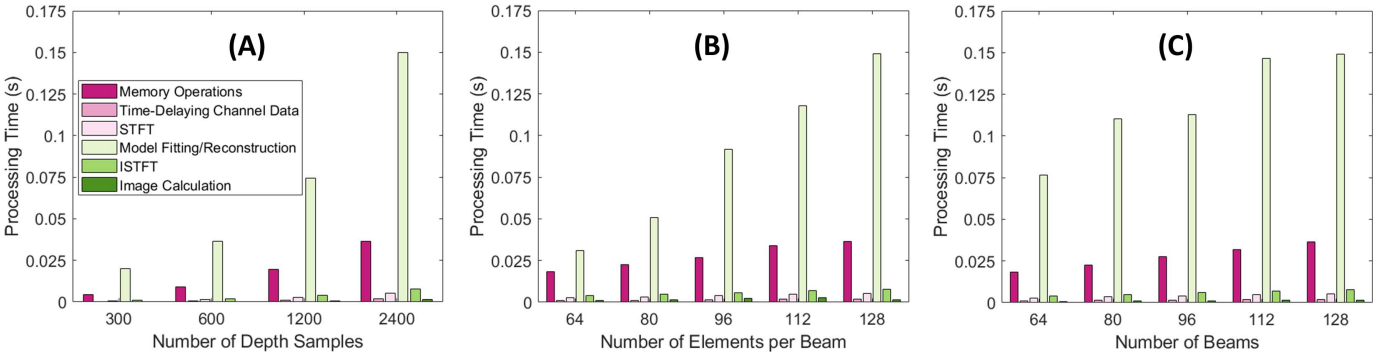

Fig. 8. 2080 Ti GPU processing times for individual stages of the GPU processing pipeline as a function of **(A)** the number of depth samples, **(B)** the number of elements per beam, and **(C)** the number of beams. When varying one parameter, the other parameters were held constant at their default values listed in Table I. Each stage has a corresponding subsection in the Methods section, and the CUDA kernels that are performed for a particular stage are the ones that are listed in its subsection. The color scheme for this figure was selected using [33].

time for the smaller data set should be reduced by a total factor of 10.5–18. Based off of the timings where the original data was processed (Figs. 6 and 7), the average processing time using the multi-GPU implementation ranged from approximately 0.168–0.173 s. Therefore, the lower bound average processing time for the smaller data set was estimated to be 0.009 s (0.168 s divided by 18), and the upper bound was estimated to be 0.016 s (0.173 s divided by 10.5). The actual processing time was determined by averaging 100 runs, and it was 0.016 s with a standard deviation of 0.001 s. This means that the performance model provided an accurate estimate of the processing time. The beam distribution for this case was that 2 beams were distributed to the 1080 Ti GPU and 62 beams were distributed to the 2080 Ti GPU.

In addition, Fig. 8 shows that the majority of the processing time for ADMIRE is spent in the model fitting and reconstruction stage. In particular, the CUDA kernel that performs the model fits requires the largest amount of computational time. Therefore, this kernel was profiled using NVIDIA's Nsight Compute profiling tool, which performs a comprehensive analysis of CUDA kernels in order to identify bottlenecks. From this analysis, it was determined that the primary bottleneck for the kernel is latency due to memory transactions. This is due to the fact that each computational thread on the

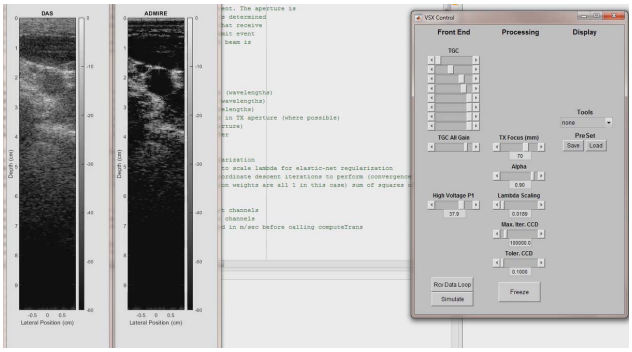

Fig. 9. Screen capture during simultaneous DAS (left) and ADMIRE (right) imaging with the L7-4 probe sequence. The images are of the carotid artery, and they are displayed with a dynamic range of 60 dB.

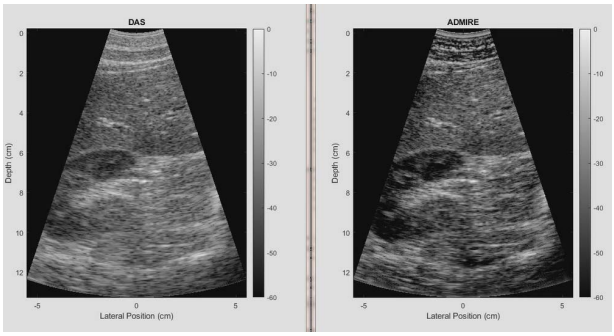

Fig. 10. Screen capture during simultaneous DAS (left) and ADMIRE (right) imaging with the C5-2 probe sequence. The images are of the liver and one kidney, and they are displayed with a dynamic range of 60 dB.

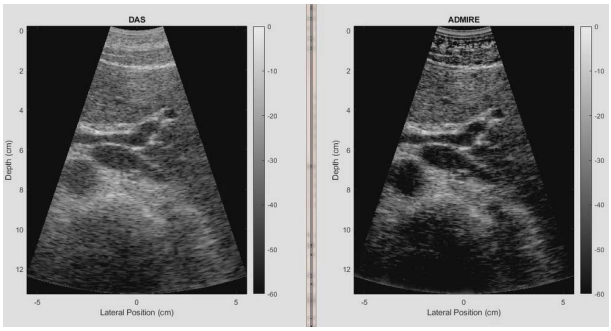

Fig. 11. Screen capture during simultaneous DAS (left) and ADMIRE (right) imaging with the C5-2 probe sequence. The images are of the liver, and they are displayed with a dynamic range of 60 dB.

GPU performs one model fit using cyclic coordinate descent, and as shown in the pseudocode for the algorithm, a large number of memory transactions is required. Essentially, if the computational threads on the GPU are waiting for memory transactions to be completed, then they cannot perform computational operations. Moreover, the number of clock cycles that these threads wait depends on if the memory being accessed is in global memory, shared memory, L2 cache memory, or L1 cache memory.

C. Verasonics Imaging

In regard to Verasonics imaging, example screen captures during imaging with each sequence are shown in Figs. 9–11. As can be seen in these figures, other beamforming techniques

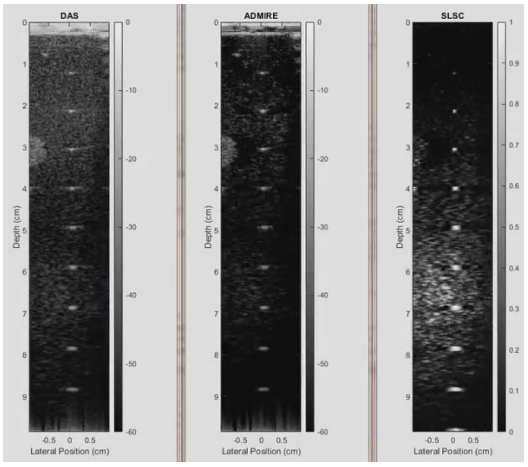

Fig. 12. Screen capture during simultaneous DAS (left), ADMIRE (center), and SLSC (right) imaging with the L7-4 probe sequence. The images are of a CIRS quality assurance phantom. The DAS and ADMIRE images are displayed with a dynamic range of 60 dB, and the SLSC image is displayed using a range from 0 to 1. Note that the SLSC lag was updated to 20 during imaging.

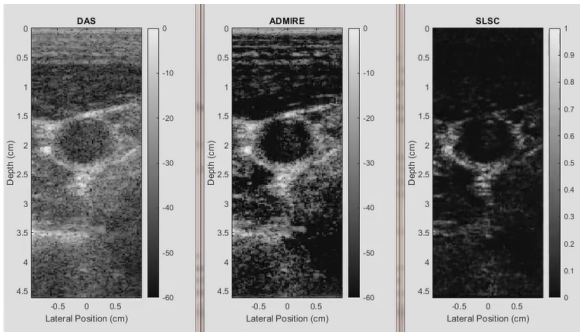

Fig. 13. Screen capture during simultaneous DAS (left), ADMIRE (center), and SLSC (right) imaging with the L7-4 probe sequence. The images are of the carotid artery. The DAS and ADMIRE images are displayed with a dynamic range of 60 dB, and the SLSC image is displayed using a range from 0 to 1.

can be computed with ADMIRE and simultaneously displayed. For example, the figures show how DAS images can also be computed on the GPU along with the ADMIRE images. This is similar to the real-time simultaneous B-mode/spatial coherence GPU-based beamformer presented in [7]–[9]. Moreover, we also demonstrate the ability to compute more than two beamforming techniques. Fig. 12 shows an example of DAS, ADMIRE, and SLSC being computed on the GPU and simultaneously displayed during imaging. This data was acquired with the L7-4 probe sequence, and a multi-purpose, multi-tissue phantom (Model 040GSE, CIRS, Norfolk, VA, USA) was scanned. Fig. 13 shows an example when using the L7-4 probe sequence to scan the carotid artery. This case used the same parameters as the other L7-4 cases, but the only difference is that it consisted of 1250 depth samples instead of 2667 depth samples. ADMIRE was applied from depth sample 25 to depth sample 1248, which means that DAS was applied to the first 24 and last two depth samples by default. Additional GUI controls were added to the Verasonics GUI in order to also change SLSC parameters.

TABLE III
MULTI-PURPOSE PHANTOM IMAGING TIMING BREAKDOWN

Processing Pipeline Stage	Processing Time (s)
Verasonics Operations	0.017
Memory Operations	0.020
DAS	0.001
ADMIRE	0.029
SLSC	0.007
DAS, ADMIRE, and SLSC Image Display	0.003-0.004

These parameters include the maximum lag between elements, the axial kernel size, and the spacing between elements that is used when a downsampled aperture is used for SLSC to improve its computational efficiency [34]. The default values for these parameters when the sequence begins are 10, 5, and 1, respectively. In regard to frame rate, it varies depending upon many factors that include the imaging and processing parameters, the number of beamforming techniques that are simultaneously displayed, and whether scan conversion is performed. For the cases presented in Figs. 9–12, the frame rate range was approximately 5–15 frames/s. The case in Fig. 10 and the case in Fig. 11 were the slowest due to the fact that scan conversion was performed, and a finely sampled cartesian grid was used. Therefore, the time for transferring and displaying the images was greater. For the case presented in Fig. 13, the frame rate was approximately 23–24 frames/s. If a parameter such as the number of beams was increased from 64 beams to a higher number in order to obtain a larger field of view, for example, then the frame rates for all of these cases would have been reduced. However, another parameter can also be changed in order to compensate for this and maintain a similar frame rate, such as decreasing the sampling frequency to allow for the depth range of the scan to be maintained while reducing the number of depth samples. Essentially, tradeoffs can be made between different parameters and the frame rate, but it is important to consider that image quality can also be impacted as a result. Videos of the cases presented in Figs. 9–13 are provided. To view them, refer to the supplementary materials for this journal article that are available at https://doi.org/10.1109/TUFFC.2021.3056334.

In addition, an example timing breakdown obtained using NVIDIA's Nsight Systems profiling tool is shown in Table III. These times are for a multi-purpose, multi-tissue phantom imaging case similar to the one shown in Fig. 12. The default ADMIRE and SLSC parameters were used except for the maximum SLSC lag, which was updated to 20. Note that the time for Verasonics operations includes all of the time for the Verasonics sequence that was not spent for image display and the external processing event that performs GPU-based processing. Moreover, the time to delay the channel data is only included in the DAS time because ADMIRE and SLSC also use the same set of delayed channel data. Therefore, the DAS time is for going from the undelayed channel data to the final DAS image before it is transferred back to the host system in order to be displayed while the ADMIRE and SLSC times are for going from the delayed channel data to the final images before being transferred back to the host system in order to be displayed. The time for the kernel that reshapes the Verasonics channel data buffer and performs data type conversion from int16 to float is not included because it was less than 1 ms. In addition, the time for image display is for the particular window sizes that are shown in Fig. 12 and its corresponding video.

IV. DISCUSSION

A. Impact of GPU Acceleration

In prior works, the ability of ADMIRE to improve ultrasound image quality has shown great promise [2]–[4], but its significant processing time requirement has not been able to be fully addressed even when a computationally efficient implementation was developed [21]. However, by utilizing GPU acceleration in this work, we have been able to significantly reduce the last remaining gap in performance that has prevented ADMIRE from achieving clinical translation. With this implementation, imaging up to several frames per second is possible. As shown in Fig. 6, when compared to the CPU implementation of ADMIRE, the GPU implementation for both of the single GPU cases and the multi-GPU implementation all provide a speedup of two orders of magnitude.

B. Effect of Imaging and Processing Parameters on Run Time

In regard to the effect of imaging and processing parameters on processing time, the number of elements per beam had a greater impact when compared to the number of depths and the number of beams. This is most likely because having fewer elements per beam results in smaller ADMIRE model matrices and aperture data sets. The effect of this is that fewer computations have to be performed per model fit. The number of beams and the number of depths affect the total number of required model fits but not the number of computations per model fit. In addition, like the number of elements per beam, the tolerance for cyclic coordinate descent, the c_λ value, and the α value affect the number of computations that need to be performed for each model fit. For example, additional or fewer iterations of cyclic coordinate descent may be required depending on the tolerance value. Moreover, increasing c_λ increases the amount of regularization, which shrinks the model coefficients and decreases the degrees of freedom in the model assuming α is greater than 0. Reducing the degrees of freedom decreases the number of computations per model fit, and this is also why increasing α to weight the L1-regularization term more results in decreased processing times. Now, besides the effects of these parameters on the computational aspects of ADMIRE, the number of depths, the number of elements per beam, and the number of beams can also affect the time taken for memory operations to occur as shown in Fig. 8. As previously stated, these operations are included in the processing times in Fig. 7, so reducing them also decreases the processing times.

In addition, for varying the number of beams, the processing time appears to change by a larger amount when the number of beams changes by a multiple of 32, as shown in Fig. 7. For example, when going from 128 beams to 112 beams, the processing time only slightly decreases, but the decrease is greater when going from 112 beams to 96 beams. This is

because, as discussed in the Results section, the majority of the time for ADMIRE is spent in the kernel that performs the model fits. This kernel uses a block size of 32, where all of the threads within one block handle performing the fits for the same row number and STFT window number across different beams. Due to this, the total number of blocks and CUDA warps required for the model fits is the same for the 128 and 112 beam cases. The 112 beam case will just have some CUDA warps where 16 threads are performing computational operations and the other 16 threads are idle, assuming that the particular GPU architecture has a warp size of 32 threads. This is why the processing time for the model fitting and reconstruction stage is similar for these two cases in Fig. 8. However, when there are 96 beams, the next multiple of 32 is reached, and the total number of required blocks and CUDA warps decreases. This also applies for going from 96 beams to 80 beams versus going from 80 beams to 64 beams.

C. Effect of GPU Hardware on Run Time

In regard to the effect of the type of GPU on processing time, Figs. 6 and 7 show that when compared to the 1080 Ti GPU, running ADMIRE on the 2080 Ti GPU is approximately 2.2 times faster. One factor that was not accounted for in these benchmarks is the fact that in the host system that contained both a 1080 Ti GPU and a 2080 Ti GPU, the 1080 Ti GPU card was configured to handle the display in addition to performing CUDA computations. This could have possibly resulted in the 1080 Ti GPU processing times being slower than they would have been if only CUDA computations were being performed. However, this result is still expected due to the hardware differences between the GPUs. For example, as previously stated, the primary performance bottleneck for the GPU implementation of ADMIRE is latency that is caused by memory transactions. Due to this, having greater amounts of low latency memory such as L1 cache memory, L2 cache memory, and shared memory will allow for improved computational performance. In the case of the 2080 Ti GPU, it has double the amount of L2 cache memory when compared to the 1080 Ti GPU. Moreover, the 1080 Ti GPU has 28 streaming multiprocessors, each with 96 KB of shared memory and 48 KB of L1 cache memory. In contrast, the 2080 Ti GPU has 68 streaming multiprocessors, each with a 96 KB unified L1/shared memory cache. Therefore, there is a significantly larger amount of low latency memory on the 2080 Ti GPU, and the fact that it has unified L1/shared memory caches instead of separate ones allows for reduced latency and improved bandwidth for L1 cache memory. In addition, the 2080 Ti GPU also has 4352 CUDA cores and a memory bandwidth of 616 GB/s while the 1080 Ti GPU has 3584 CUDA cores and a memory bandwidth of 484 GB/s. Having more CUDA cores means that there are more computational units available to perform calculations, and having a higher memory bandwidth helps provide all of these units with data to process.

Now, in regard to the multi-GPU implementation, it is approximately 2.6 times faster than the 1080 Ti GPU and approximately 1.2 times faster than the 2080 Ti GPU. This is important because all of the Verasonics imaging was performed using a single 2080 Ti GPU. The frame rates ranged from 5 to 15 frames/s for the cases presented in Figs. 9–12, and the frame rate was approximately 23–24 frames/s for the case presented in Fig. 13. All of these cases involved imaging with more than one beamforming method at a time, but ADMIRE was the primary computational bottleneck instead of DAS or SLSC as can be seen in Table III. These frame rates were adequate for these cases, but higher frame rates might be desired for cases such as cardiac imaging. Therefore, for cases like this, the multi-GPU implementation of ADMIRE could be used to provide higher frame rates, and DAS and SLSC could also be incorporated into the multi-GPU implementation for simultaneous imaging like they were incorporated into the single GPU implementation. For example, if imaging were to be performed using the multi-GPU implementation with two 2080 Ti GPUs instead of one, then we would expect the frame rates for all of the presented cases to double. This is due to the fact that the processing time for ADMIRE decreases linearly, approximately, as the number of beams also decreases, and distributing half of the beams to one GPU and the other half to the other GPU results in the beams per GPU being reduced by half. Note that in our presented benchmarks, we stated that a reduction factor of 2 in the number of beams results in the processing time being reduced by a factor of 1.5–2. The lower end was for the multi-GPU implementation, and this is most likely due to the hardware differences between the GPUs, as discussed above.

V. Conclusion

We have developed a GPU implementation of ADMIRE that is two orders of magnitude faster than the CPU implementation, and additional speedup is achieved when using multiple GPUs. Moreover, we have demonstrated the feasibility of real-time imaging with ADMIRE. We have also shown that other beamforming techniques such as DAS and SLSC can be computed on the GPU and simultaneously displayed with ADMIRE up to several frames per second. Future work includes making further optimizations to the imaging pipeline. For example, two optimizations are to use OpenGL in order to display images on the GPU without having to transfer them back to the host system first and to implement image scan conversion using texture memory in order to perform fast bilinear interpolation. Another optimization is to analyze the GPU kernels in order to see if memory access efficiency can be improved further. This includes factors such as increasing coalesced memory access and better utilizing memory resources such as shared memory.

Appendix
Derivation of the Coordinate Descent Update for Linear Regression With Elastic-Net Regularization

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \frac{1}{2N} \sum_{i=1}^{N} \left(\boldsymbol{y}_i - \sum_{j=1}^{P} \boldsymbol{X}_{ij} \boldsymbol{\beta}_j \right)^2 + \lambda \left(\alpha \|\boldsymbol{\beta}\|_1 + \frac{(1-\alpha)\|\boldsymbol{\beta}\|_2^2}{2} \right) \quad (1)$$

Compute the subgradient of the objective function f with respect to one model coefficient at a time.

$$\partial_{\boldsymbol{\beta}_j} f = \frac{1}{N}\sum_{i=1}^{N}\left(\left(\boldsymbol{y}_i - \sum_{k=1}^{P}\boldsymbol{X}_{ik}\boldsymbol{\beta}_k\right)(-\boldsymbol{X}_{ij})\right) + \lambda\alpha\partial_{\boldsymbol{\beta}_j}|\boldsymbol{\beta}_j| + \lambda(1-\alpha)\boldsymbol{\beta}_j \quad (2)$$

$$\partial_{\boldsymbol{\beta}_j} f = -\frac{1}{N}\sum_{i=1}^{N}\left(\boldsymbol{X}_{ij}\left(\boldsymbol{y}_i - \sum_{k\neq j}^{P}\boldsymbol{X}_{ik}\boldsymbol{\beta}_k - \boldsymbol{X}_{ij}\boldsymbol{\beta}_j\right)\right) + \lambda\alpha\partial_{\boldsymbol{\beta}_j}|\boldsymbol{\beta}_j| + \lambda(1-\alpha)\boldsymbol{\beta}_j \quad (3)$$

$$\partial_{\boldsymbol{\beta}_j} f = -\frac{1}{N}\sum_{i=1}^{N}\boldsymbol{X}_{ij}\left(\boldsymbol{y}_i - \sum_{k\neq j}^{P}\boldsymbol{X}_{ik}\boldsymbol{\beta}_k\right) + \frac{1}{N}\sum_{i=1}^{N}\boldsymbol{X}_{ij}^2\boldsymbol{\beta}_j + \lambda\alpha\partial_{\boldsymbol{\beta}_j}|\boldsymbol{\beta}_j| + \lambda(1-\alpha)\boldsymbol{\beta}_j \quad (4)$$

For convenience, we can substitute ρ_j for $\sum_{i=1}^{N}\boldsymbol{X}_{ij}(\boldsymbol{y}_i - \sum_{k\neq j}^{P}\boldsymbol{X}_{ik}\boldsymbol{\beta}_k)$. Moreover, we can substitute 1 for $\sum_{i=1}^{N}\boldsymbol{X}_{ij}^2$ (1 can be substituted due to predictor normalization).

$$\partial_{\boldsymbol{\beta}_j} f = -\frac{1}{N}\rho_j + \frac{1}{N}\boldsymbol{\beta}_j + \lambda\alpha\partial_{\boldsymbol{\beta}_j}|\boldsymbol{\beta}_j| + \lambda(1-\alpha)\boldsymbol{\beta}_j \quad (5)$$

Set the subgradient equal to 0 in order to find the minimizer.

$$-\frac{1}{N}\rho_j + \frac{1}{N}\boldsymbol{\beta}_j + \lambda\alpha\partial_{\boldsymbol{\beta}_j}|\boldsymbol{\beta}_j| + \lambda(1-\alpha)\boldsymbol{\beta}_j = 0 \quad (6)$$

Find the subgradient given by $\lambda\alpha\partial_{\boldsymbol{\beta}_j}|\boldsymbol{\beta}_j|$.

$$\lambda\alpha\partial_{\boldsymbol{\beta}_j}|\boldsymbol{\beta}_j| = \begin{cases} -\lambda\alpha, & \text{if } \boldsymbol{\beta}_j < 0 \\ [-\lambda\alpha, \lambda\alpha], & \text{if } \boldsymbol{\beta}_j = 0 \\ \lambda\alpha, & \text{if } \boldsymbol{\beta}_j > 0 \end{cases}$$

Find the minimizer for each case.

Case 1 ($\beta_j < 0$):

$$-\frac{1}{N}\rho_j + \frac{1}{N}\boldsymbol{\beta}_j - \lambda\alpha + \lambda(1-\alpha)\boldsymbol{\beta}_j = 0$$
$$-\frac{1}{N}\rho_j + \frac{1}{N}\boldsymbol{\beta}_j - \lambda\alpha + \lambda\boldsymbol{\beta}_j - \lambda\alpha\boldsymbol{\beta}_j = 0$$
$$\frac{1}{N}\boldsymbol{\beta}_j + \lambda\boldsymbol{\beta}_j - \lambda\alpha\boldsymbol{\beta}_j = \frac{1}{N}\rho_j + \lambda\alpha$$

$\boldsymbol{\beta}_j = ((1/N)\rho_j + \lambda\alpha)/((1/N) + \lambda(1-\alpha))$ if $(1/N)\rho_j < -\lambda\alpha$ (this is when $\boldsymbol{\beta}_j < 0$).

Case 2 ($\beta_j = 0$): In order for $\boldsymbol{\beta}_j = 0$ to be an optimum, we need $[-(1/N)\rho_j - \lambda\alpha, -(1/N)\rho_j + \lambda\alpha]$ to contain 0, which means that

$$-\frac{1}{N}\rho_j - \lambda\alpha \leq 0 \quad \text{and} \quad -\frac{1}{N}\rho_j + \lambda\alpha \geq 0$$
$$\boldsymbol{\beta}_j = 0 \quad \text{if} \quad -\lambda\alpha \leq \frac{1}{N}\rho_j \leq \lambda\alpha.$$

Case 3 ($\beta_j > 0$):

$$-\frac{1}{N}\rho_j + \frac{1}{N}\boldsymbol{\beta}_j + \lambda\alpha + \lambda(1-\alpha)\boldsymbol{\beta}_j = 0$$
$$-\frac{1}{N}\rho_j + \frac{1}{N}\boldsymbol{\beta}_j + \lambda\alpha + \lambda\boldsymbol{\beta}_j - \lambda\alpha\boldsymbol{\beta}_j = 0$$
$$\frac{1}{N}\boldsymbol{\beta}_j + \lambda\boldsymbol{\beta}_j - \lambda\alpha\boldsymbol{\beta}_j = \frac{1}{N}\rho_j - \lambda\alpha$$

$\boldsymbol{\beta}_j = ((1/N)\rho_j - \lambda\alpha)/((1/N) + \lambda(1-\alpha))$ if $(1/N)\rho_j > \lambda\alpha$ (this is when $\boldsymbol{\beta}_j > 0$).

These three cases provide the following coordinate descent update

$$\hat{\boldsymbol{\beta}}_j \leftarrow \frac{S\left(\frac{1}{N}\rho_j, \lambda\alpha\right)}{\frac{1}{N} + \lambda(1-\alpha)}$$
$$S(z,\gamma) = \begin{cases} z-\gamma, & \text{if } z > 0 \text{ and } \gamma < |z| \\ z+\gamma, & \text{if } z < 0 \text{ and } \gamma < |z| \\ 0, & \text{if } \gamma \geq |z|. \end{cases} \quad (7)$$

We can substitute $\sum_{i=1}^{N}\boldsymbol{X}_{ij}(\boldsymbol{y}_i - \sum_{k\neq j}^{P}\boldsymbol{X}_{ik}\boldsymbol{\beta}_k)$ back in for ρ_j to obtain

$$\hat{\boldsymbol{\beta}}_j \leftarrow \frac{S\left(\frac{1}{N}\sum_{i=1}^{N}\boldsymbol{X}_{ij}\left(\boldsymbol{y}_i - \sum_{k\neq j}^{P}\boldsymbol{X}_{ik}\boldsymbol{\beta}_k\right), \lambda\alpha\right)}{\frac{1}{N} + \lambda(1-\alpha)}. \quad (8)$$

We can then substitute $\hat{\boldsymbol{y}}_i^{(j)}$ for $\sum_{k\neq j}^{P}\boldsymbol{X}_{ik}\boldsymbol{\beta}_k$ to obtain the coordinate descent update in (2) of the Methods section, which is

$$\hat{\boldsymbol{\beta}}_j \leftarrow \frac{S\left(\frac{1}{N}\sum_{i=1}^{N}\boldsymbol{X}_{ij}\left(\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i^{(j)}\right), \lambda\alpha\right)}{\frac{1}{N} + \lambda(1-\alpha)}$$
$$S(z,\gamma) = \begin{cases} z-\gamma, & \text{if } z > 0 \text{ and } \gamma < |z| \\ z+\gamma, & \text{if } z < 0 \text{ and } \gamma < |z| \\ 0, & \text{if } \gamma \geq |z|. \end{cases} \quad (9)$$

ACKNOWLEDGMENT

The authors would like to thank the staff of the Vanderbilt University ACCRE computing resource. They would also like to thank the reviewers of this journal article for their helpful comments and suggestions.

REFERENCES

[1] G. F. Pinton, G. E. Trahey, and J. J. Dahl, "Sources of image degradation in fundamental and harmonic ultrasound imaging using nonlinear, full-wave simulations," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 58, no. 4, pp. 754–765, Apr. 2011.

[2] B. Byram and M. Jakovljevic, "Ultrasonic multipath and beamforming clutter reduction: A chirp model approach," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 61, no. 3, pp. 428–440, Mar. 2014.

[3] B. Byram, K. Dei, J. Tierney, and D. Dumont, "A model and regularization scheme for ultrasonic beamforming clutter reduction," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 62, no. 11, pp. 1913–1927, Nov. 2015.

[4] K. Dei and B. Byram, "The impact of model-based clutter suppression on cluttered, aberrated wavefronts," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 64, no. 10, pp. 1450–1464, Oct. 2017.

[5] B. Dsp, "GPU vs FPGA performance comparison," White Paper, 2016. [Online]. Available: https://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf

[6] M. A. Lediju, G. E. Trahey, B. C. Byram, and J. J. Dahl, "Short-lag spatial coherence of backscattered echoes: Imaging characteristics," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 58, no. 7, pp. 1377–1388, Jul. 2011.

[7] D. Hyun, G. Trahey, and J. Dahl, "A GPU-based real-time spatial coherence imaging system," *Proc. SPIE*, vol. 8675, Mar. 2013, Art. no. 86751B.

[8] D. Hyun, G. E. Trahey, and J. J. Dahl, "*In vivo* demonstration of a real-time simultaneous B-mode/spatial coherence GPU-based beamformer," in *Proc. IEEE Int. Ultrason. Symp. (IUS)*, Jul. 2013, pp. 1280–1283.

[9] D. Hyun, G. E. Trahey, and J. J. Dahl, "Real-time high-framerate *in vivo* cardiac SLSC imaging with a GPU-based beamformer," in *Proc. IEEE Int. Ultrason. Symp. (IUS)*, Oct. 2015, pp. 1280–1283.

[10] I. K. Holfort, F. Gran, and J. A. Jensen, "Broadband minimum variance beamforming for ultrasound imaging," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 56, no. 2, pp. 314–325, Feb. 2009.

[11] J. F. Synnevag, A. Austeng, and S. Holm, "Adaptive beamforming applied to medical ultrasound imaging," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 54, no. 8, pp. 1606–1613, Aug. 2007.

[12] J. P. Asen, J. I. Buskenes, C.-I.-C. Nilsen, A. Austeng, and S. Holm, "Implementing capon beamforming on a GPU for real-time cardiac ultrasound imaging," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 61, no. 1, pp. 76–85, Jan. 2014.

[13] B. Y. S. Yiu and A. C. H. Yu, "GPU-based minimum variance beamformer for synthetic aperture imaging of the eye," *Ultrasound Med. Biol.*, vol. 41, no. 3, pp. 871–883, Mar. 2015.

[14] B. Y. S. Yiu, I. K. H. Tsang, and A. C. H. Yu, "GPU-based beamformer: Fast realization of plane wave compounding and synthetic aperture imaging," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 58, no. 8, pp. 1698–1705, Aug. 2011.

[15] S. Rosenzweig, M. Palmeri, and K. Nightingale, "GPU-based real-time small displacement estimation with ultrasound," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 58, no. 2, pp. 399–405, Feb. 2011.

[16] Y. Dai, J. Tian, D. Dong, G. Yan, and H. Zheng, "Real-time visualized freehand 3D ultrasound reconstruction based on GPU," *IEEE Trans. Inf. Technol. Biomed.*, vol. 14, no. 6, pp. 1338–1345, Nov. 2010.

[17] A. J. Y. Chee, B. Y. S. Yiu, and A. C. H. Yu, "A GPU-parallelized eigen-based clutter filter framework for ultrasound color flow imaging," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 64, no. 1, pp. 150–163, Jan. 2017.

[18] D. Hyun, L. L. Brickson, K. T. Looby, and J. J. Dahl, "Beamforming and speckle reduction using neural networks," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 66, no. 5, pp. 898–910, May 2019.

[19] R. Göbl, N. Navab, and C. Hennersperger, "SUPRA: Open-source software-defined ultrasound processing for real-time applications," *Int. J. Comput. Assist. Radiol. Surg.*, vol. 13, no. 6, pp. 759–767, Jun. 2018.

[20] D. Hyun, Y. L. Li, I. Steinberg, M. Jakovljevic, T. Klap, and J. J. Dahl, "An open source GPU-based beamformer for real-time ultrasound imaging and applications," in *Proc. IEEE Int. Ultrason. Symp. (IUS)*, Oct. 2019, pp. 20–23.

[21] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *J. Roy. Stat. Soc. B, Stat. Methodol.*, vol. 67, no. 2, pp. 301–320, Apr. 2005.

[22] K. Dei, S. Schlunk, and B. Byram, "Computationally efficient implementation of aperture domain model image reconstruction," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 66, no. 10, pp. 1546–1559, Oct. 2019.

[23] J. F. Cardoso, "Source separation using higher order moments," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, 1989, pp. 2109–2112.

[24] J. Shlens, "A tutorial on independent component analysis," 2014, *arXiv:1404.2986*. [Online]. Available: http://arxiv.org/abs/1404.2986

[25] C. Khan and B. Byram, "GENRE (GPU Elastic-Net REgression): A CUDA-accelerated package for massively parallel linear regression with elastic-net regularization," *J. Open Source Softw.*, vol. 5, no. 54, p. 2664, 2020.

[26] J. Friedman, T. Hastie, and R. Tibshirani, "Regularization paths for generalized linear models via coordinate descent," *J. Stat. Softw.*, vol. 33, no. 1, pp. 1–22, 2010.

[27] J. Qian, T. Hastie, J. Friedman, R. Tibshirani, and N. Simon. (2013). *Glmnet for MATLAB*. [Online]. Available: http://www.stanford.edu/~hastie/glmnet_MATLAB/

[28] T. Hastie and J. Qian. (2014). *Glmnet Vignette*. Accessed: Sep. 20, 2016. [Online]. Available: http://www.web.stanford.edu~hastie/Papers/Glmnet_Vignette.pdf

[29] M. Harris, "Optimizing parallel reduction in CUDA," *Nvidia Developer Technol.*, vol. 2, no. 4, p. 70, 2007.

[30] L. Marple, "Computing the discrete-time 'analytic' signal via FFT," *IEEE Trans. Signal Process.*, vol. 47, no. 9, pp. 2600–2603, Sep. 1999.

[31] J. A. Jensen, "FIELD: A program for simulating ultrasound systems," in *Proc. 10th Nordic-Baltic Conf. Biomed. Imag.*, 1996, vol. 4, no. 1, pp. 351–353.

[32] J. A. Jensen and N. B. Svendsen, "Calculation of pressure fields from arbitrarily shaped, apodized, and excited ultrasound transducers," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 39, no. 2, pp. 262–267, Mar. 1992.

[33] C. A. Brewer, M. Harrower, B. Sheesley, A. Woodruff, and D. Heyman. (2013). *ColorBrewer 2.0*. Accessed: Jan. 30, 2015. [Online]. Available: https://colorbrewer2.org

[34] D. Hyun, A. L. C. Crowley, and J. J. Dahl, "Efficient strategies for estimating the spatial coherence of backscatter," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 64, no. 3, pp. 500–513, Mar. 2017.

Christopher Khan (Student Member, IEEE) was born in Nashville, TN, USA. He received the B.E. degree in biomedical engineering with a minor in scientific computing from Vanderbilt University, Nashville, TN, USA, in 2018, where he is currently pursuing the Ph.D. degree in biomedical engineering with the Biomedical Elasticity and Acoustic Measurement (BEAM) Laboratory.

He is also affiliated with the Vanderbilt Institute for Surgery and Engineering (VISE). His current research interests include signal processing, machine learning, and developing real-time implementations of ultrasound beamforming techniques.

Kazuyuki Dei was born and raised in Saitama, Japan. He received the B.S. degree in mechanical engineering from the University of California, Berkeley, CA, USA, and the Ph.D. degree in biomedical engineering from Vanderbilt University, Nashville, TN, USA, in 2000 and 2019, respectively. He worked for Fujitsu Limited and Fujitsu Semiconductor America Inc. as an electrical design and development engineer for more than 12 years. He is currently a lead ultrasound systems engineer at GE Healthcare, Wauwatosa, WI, USA.

Siegfried Schlunk (Student Member, IEEE) was born in Nashville, TN, USA. He received the B.E. degree in biomedical engineering and mathematics from Vanderbilt University, Nashville, TN, USA, in 2016, where he is currently pursuing the Ph.D. degree in biomedical engineering.

He is also affiliated with the Vanderbilt Institute for Surgery and Engineering (VISE). His research interests focus on developing methods for improving ultrasound image quality in cardiac and kidney applications.

Kathryn Ozgun (Student Member, IEEE) received the B.S. degree in biomedical engineering from North Carolina State University, Raleigh, NC, USA, in 2016. She is currently pursuing the Ph.D. degree in biomedical engineering at Vanderbilt University, Nashville, TN, USA.

She is currently a Russell G. Hamilton Scholar and holds a Graduate Certificate in surgical and interventional engineering. She is also affiliated with the the Vanderbilt Institute for Surgery and Engineering (VISE) and the Vanderbilt University Institute of Imaging Science (VUIIS). Her research is focused on coherence-based beamforming and adaptive filtering methods for ultrasound blood flow imaging.

Brett Byram (Member, IEEE) received the B.S. degree in biomedical engineering and math from Vanderbilt University, Nashville, TN, USA, in 2004, and the Ph.D. degree in biomedical engineering from Duke University, Durham, NC, USA, in 2011.

He was a Research Assistant Professor with Duke University. In 2013, he joined the Department of Biomedical Engineering, Vanderbilt University, where he is currently an Associate Professor. He has spent time working in Jørgen Jensen's Center for Fast Ultrasound, Kongens Lyngby, Denmark, and the Ultrasound Division, Siemens Healthcare, Mountain View, CA, USA. He currently runs the Biomedical Elasticity and Acoustic Measurement (BEAM) Laboratory, where he and others in the lab pursue solutions to clinical problems using ultrasound. He is also affiliated with the Vanderbilt Institute for Surgery and Engineering (VISE) and the Vanderbilt University Institute of Imaging Science (VUIIS). His research interests include beamforming, motion estimation, and other related signal processing and hardware development tasks.