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

An Open Source GPU-Based Beamformer for Real-Time Ultrasound Imaging and Applications

Dongwoon Hyun*, You Leo Li*, Idan Steinberg*, Marko Jakovljevic*, Tal Klap[†], and Jeremy J. Dahl*
*Department of Radiology, Stanford University, Stanford, CA 94305, [†]Independent
Email: dongwoon.hyun@stanford.edu

Abstract—**Recent technological advances in graphics processing unit (GPU)-based computing have made it possible to visualize customized beamforming pipelines and algorithms in real-time. However, GPU programming is challenging and poses a significant barrier to its widespread adoption in the ultrasound research community. Here, we present an open source GPU beamformer with the intent of making GPU beamforming more accessible to a wider audience. The beamformer was written in C++/CUDA and is comprised of a library of core classes to perform typical ultrasound-related tasks, such as applying focusing delays. Classes are arranged into a computational graph that is fixed at compile-time, enabling high throughput at runtime. Concrete examples are provided to demonstrate how to interface the beamforming library with the MATLAB-based Verasonics platform to perform live B-mode and Doppler imaging. Also provided is an example of deploying a TensorFlow neural network in real-time via TensorRT. Compilation was performed using CMake to allow for cross-platform compatibility. Including overhead for data acquisition, the beamformer achieved live B-mode imaging with a Verasonics Vantage 256 system at 55 frames per second using a single NVIDIA Titan V GPU. The open source GPU beamformer can be used as a starting point for real-time algorithm deployment.**

I. Introduction

Recently, open platform ultrasound systems have significantly expanded the capabilities of academic ultrasound researchers to prototype and test new image reconstruction techniques. These systems allow customization of the transmit waveform, provide access to the pre-beamformed data, and can be used for real-time imaging [1]. In medical ultrasound imaging, these systems have drastically reduced the time from conception of an idea to its real-time manifestation for presentation to clinicians [2], accelerating the rate of research. Although technological advances have made real-time custom imaging feasible, implementation is still challenging.

Pulse-echo ultrasound image reconstruction is inherently a parallel computing task. The backscatter from a point in the field is often considered to depend only on a small neighborhood around the field point (e.g., the point spread function), and thus backscatter properties such as magnitude and frequency shift are considered to be *local* to the field point. Consequently, many pulse-echo image reconstruction algorithms (e.g., B-mode, displacement estimation) are formulated to be independent of the field point position, i.e., the same algorithm is applied to multiple field points. This type of parallel computation is classified in computer science as single instruction, multiple data (SIMD).

Graphics processing units (GPUs) are designed for parallel computing tasks and thus are well-suited for ultrasound image reconstruction. GPUs are specialized processors consisting of many cores. Although GPU cores are individually weaker than general-purpose central processing units (CPUs), their SIMD architecture makes them significantly more efficient and powerful for parallelizable tasks. GPUs have been used extensively to accelerate ultrasound computations, both in offline applications and real-time deployment [2]–[8], including an open source implementation called SUPRA [9].

Academic research laboratories in ultrasound imaging do not commonly employ software engineers, and thus GPU computing requires a significant investment of time and effort. The SIMD nature of GPU programming is fundamentally different from traditional programming, and can require some time to become acclimated. Even after an algorithm has been optimized in a testbed, the minutiae of compiling and executing an algorithm on the target system (e.g., debugging compiler and linker errors) can be surprisingly time-consuming.

The intent of this work is to alleviate this burden and make GPU beamforming accessible to a wider audience by sharing the framework used in our previous work [2]. Emphasis is placed on readability, usability, and code structure rather than on computational and/or memory efficiency. Even so, we show that the code easily exceeds 50 frames per second for live B-mode imaging. Below, we describe the code architecture and provide simple examples to show how the code can be utilized for real-time imaging on a Verasonics imaging system.

II. Software Beamformer Architecture

The GPU beamformer, referred to as `rtbf` (real-time beamforming), is open source and is available as a git repository at https://gitlab.com/dongwoon.hyun/rtbf. The `rtbf` repository consists of a central library called `gpuBF`, as well as auxiliary libraries `annBF` and `vsxBF` to demonstrate the interface with deep learning libraries and a Verasonics research scanner. The core `gpuBF` library is written in C++ using the NVIDIA CUDA application programming interface (API). Installation instructions are provided at the repository webpage. CMake was used as a cross-platform compiler to simplify the compilation and setup. CMake generates and issues the actual compilation commands on the user's respective platform (e.g., Windows, Mac, Linux), requiring the user only to specify the installation locations of the prerequisite software (e.g., CUDA, MATLAB).

A. Data Management

Data management is essential in GPU computing. Currently, a major bottleneck in GPU computing is the transfer of data between CPU memory and GPU memory. Hence, the ideal workflow consists of a single CPU to GPU data transfer at the beginning, followed by computations on the GPU using GPU memory, followed by a final transfer of the processed result from GPU to CPU memory (if not displayed directly from the GPU).

A data management class called **DataArray** is included in **gpuBF**. Upon initialization, a **DataArray** allocates GPU memory of the requested size and keeps track of a pointer to the memory. **DataArray**s maintain an internal representation of the data dimensions, which are stored as (from fastest changing to slowest changing): rows, columns, channels, frames. Pointers to **DataArray**s can be passed between **DataProcessor**s (described below) without the need to actually move data around, increasing throughput. The **DataArray** class also provides functions for synchronous and asynchronous memory transfer to and from the GPU. **DataArray** is a class template and thus can be used to store various datatypes such as **short** and **float2**. Upon cleanup, each **DataArray** is responsible for freeing the memory it has allocated.

B. Data Processing

DataArrays are processed by classes derived from an abstract base class called **DataProcessor** that enforces the overall structure of a computational graph. Each **DataProcessor** accepts a pointer to a **DataArray**, processes the input data, and produces a pointer to an output **DataArray**. Although a new **DataArray** is generated by default, the class can also be modified to operate in place as well. The actual processing classes specify their own internal processing functions in the form of CUDA kernels. Examples classes include:

- **HilbertTransform** Converts a real signal into its analytic signal via the Hilbert transform.
- **Focus** Applies focusing time delays as specified by user, utilizing texture hardware to perform linear interpolation.
- **FocusSynAp** Applies synthetic aperture focusing using coherent summation of either the transmit (default) or receive aperture.
- **ChannelSum** Decimates the channels into a specified number of channels (1 output channel by default).
- **Bmode** Detects envelope and optionally applies logarithmic compression. Performs incoherent compounding if more than one input channel is provided.
- **EnsembleFilter** Applies a "slow-time" user-specified FIR filter across the ensemble dimension.
- **PowerEstimator** Accumulates the measured power over the ensemble dimension.

C. Computational Graph

DataProcessors are arranged into a computational graph by passing **DataArray**s from one to the next. Utilizing a computational graph is a two-step process: initialization and real-time execution. All time-consuming memory allocations are performed and pre-computable values obtained during the initialization step. Once initialized, the graph can be executed repeatedly in real-time by streaming in new data.

Listing 1. Initialization of Computational Graph

```
1  // Declare DataProcessor objects
2  DataArray<float2> DA;            // datatype float2
3  HilbertTransform<float2> HT;     // in-place
4  ChannelSum<float2, float2> CS; // outputs float2
5  Bmode<float2, float> B;          // outputs float
6  // Initialize computational graph
7  DA.initialize(nrows, ncols, nchans, nframes);
8  HT.initialize(&DA); // DA --> HT
9  CS.initialize(&HT); // HT --> CS
10 B.initialize(&CS);  // CS --> B
```

Listing 2. Real-Time Execution

```
1 // Copy data from float2 *h_input in CPU memory
2 DA.copyToGPU(h_input);
3 HT.applyHilbertTransform();
4 CS.sumChannels();
5 B.detectEnvelopeLogCompress();
6 // Copy data into float *h_output in CPU memory
7 B.getOutputDataArray()->copyFromGPU(h_output);
```

For example, Listings 1 and 2 provide example code snippets for a graph that takes an input **DataArray** of **float2** datatype, applies a Hilbert transformation, sums the channels, and then detects the envelope and applies logarithmic compression. Lines 2–5 declare the objects and their input and output datatypes. Line 7 shows the initialization of a **DataArray**. Lines 8–10 in Listing 1 show how the computational graph is linked together using the various **DataProcessor**s. By passing the pointer to the previous **DataProcessor** as input, its output **DataArray** is automatically linked as the input to the next **DataProcessor**. In this example, no extra parameters were needed to initialize the objects. More sophisticated classes such as **Focus** require additional information (e.g., delay and apodization tables) for initialization.

Lines 2–5 in Listing 2 show a typical real-time execution function. First, the data is copied from CPU memory into the **DataArray**'s GPU memory. One by one, the objects execute their core functions, each of which calls one or more CUDA kernels or CUDA libraries, such as cuFFT. Finally, the output of the final class is copied back into CPU memory. In actual real-time applications, the **copyToGPU** calls could be replaced by **copyToGPUAsync** calls with an additional **cudaStream_t** input to enable execution asynchronously from the host thread.

D. Neural Network Processing in Python

Due to the recent popularity of deep learning, major software companies have devoted significant effort into developing open source environments for quick and efficient tensor operations. The tensor operations used in deep learning can also be used for ultrasound imaging. For instance, the task of channel summation can be viewed as a "convolution" about the image pixel dimensions using a 1×1 filter of ones in the channel dimension, such that the output is simply the channel sum.

One could conceivably construct an ultrasound beamformer as a "neural network" composed of tensor computations using deep learning frameworks such as TensorFlow or PyTorch. Two such examples include [10]. This approach is advantageous because the beamformers can be programmed in the more user-friendly Python language, and can take advantage of any built-in optimizations for GPU execution that are often available. Moreover, hardware companies including NVIDIA are actively developing APIs such as TensorRT to facilitate the real-time deployment of neural networks implemented in these frameworks on their GPU hardware. Although some of these tools are still in their infancy, we foresee a future in which researchers will implement their beamforming algorithms as tensor operations in high-level Python and rapidly deploy them in real-time using software that is already optimized for the target hardware, with minimal need for manually writing and tuning computation kernels.

As such, we include in **annBF** an example of channel sum and envelope detection, written as an artificial neural network. The neural network is written using TensorFlow Keras and stored in a file format that can be interpreted by TensorRT. A **NeuralNetwork** class (derived from **DataProcessor**) is provided in **annBF** to construct the neural network on the GPU and to stream data from the input **DataArray** into the network in real-time. We additionally include the network used to perform speckle-reduced B-mode imaging [11] available at https://gitlab.com/dongwoon.hyun/nn_bmode/.

E. Unit Testing

To adhere to common software development best practices, we also include unit testing for some of the classes within **gpuBF**. Unit testing was performed using the **googletest** C++ testing framework. The framework is automatically downloaded and set up by CMake for use with **rtbf**. The unit tests consist of simple examples that test the function of each **DataProcessor**, covering edge cases when possible.

III. Application to Real-Time Imaging

We present several example imaging scripts from **vsxBF** for live imaging on the Verasonics Vantage research platform.

A. Data Formatting

For convenience, a **VSXDataFormatter** class is provided in **vsxBF** to convert the raw channel data buffer into the unfocused analytic signal. On the Verasonics platform, sampling can be performed either as baseband in-phase and quadrature (IQ) data, or more redundantly, as modulated radiofrequency (RF) data. For IQ data, **VSXDataFormatter** applies the necessary interpolation to align the quadrature data, and for RF data, **VSXDataFormatter** applies a **HilbertTransform** to obtain the quadrature data. Additionally, **VSXDataFormatter** applies the proper reshaping and dimensional transposition to ignore the extra samples at the end of the raw buffer and to orient the data for interfacing with **gpuBF**. Other utilities are also included, such as copying data to and retrieving data from MATLAB **mxArray**s.

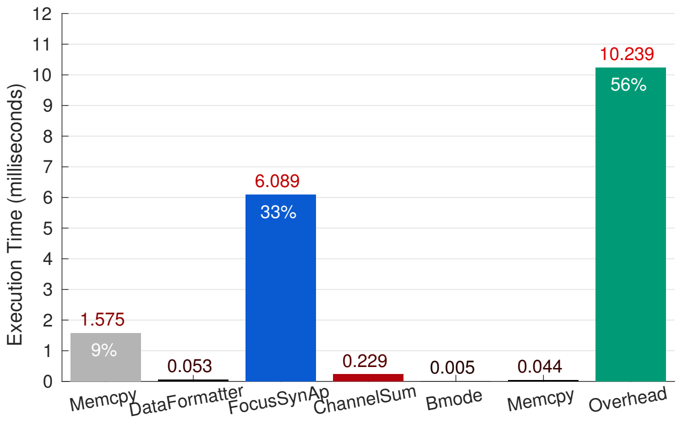

Fig. 1. The execution time of the computational graph during the real-time loop is shown, along with memory copies to and from the device and the overhead time. Overhead constituted 56% of the total time, followed by synthetic aperture focusing (33%) and copying raw data to the GPU (9%).

B. Example: B-mode Imaging

An example script of B-mode imaging with plane wave compounding is included in **vsxBF**. Our setup consisted of a Verasonics Vantage 256 system connected to a Linux workstation housing an NVIDIA Titan V GPU. An L12-3v transducer with a center frequency of 8 MHz was used to transmit 25 plane waves at angles ranging from -5° to +5°. The output image pixel grid was 30 mm in depth and 20 mm in azimuth sampled at a spacing of 3 samples per wavelength (i.e., sound speed divided by center frequency), corresponding to 456×305 pixels. The computational graph consisted of the following **DataProcessor**s:

1) **VSXDataFormatter<short,float2>**
2) **FocusSynAp<float2,float2>**
3) **ChannelSum<float2,float2>**
4) **Bmode<float2,float>**

Real-time execution of the computational graph is profiled in Fig. 1. The overhead (time from the end of computation of one frame to the beginning of the next frame) dominated the time, contributing to 56% of the total time. Note that overhead includes the transfer of the raw buffer from the Verasonics to the workstation and any CPU overhead in executing the code. Synthetic aperture focusing of 25 plane waves took approximately 33% of the total time. Transfer of the raw unprocessed data from CPU to GPU memory accounted for 9% of the total time. Channel summation, data formatting, envelope detection and logarithmic compression, and memory copy of the processed B-mode image were negligible. Excluding the overhead, the theoretical maximum computational throughput was 125 frames per second. Including overhead, the actual frame rate was 55 frames per second.

A screenshot of the real-time display is shown in Fig. 2 for a different B-mode acquisition (40 mm by 25 mm field of view). Imaging was performed in a phantom containing cylindrical anechoic lesions of various diameters. In this imaging case, live imaging was performed at 47 frames per second.

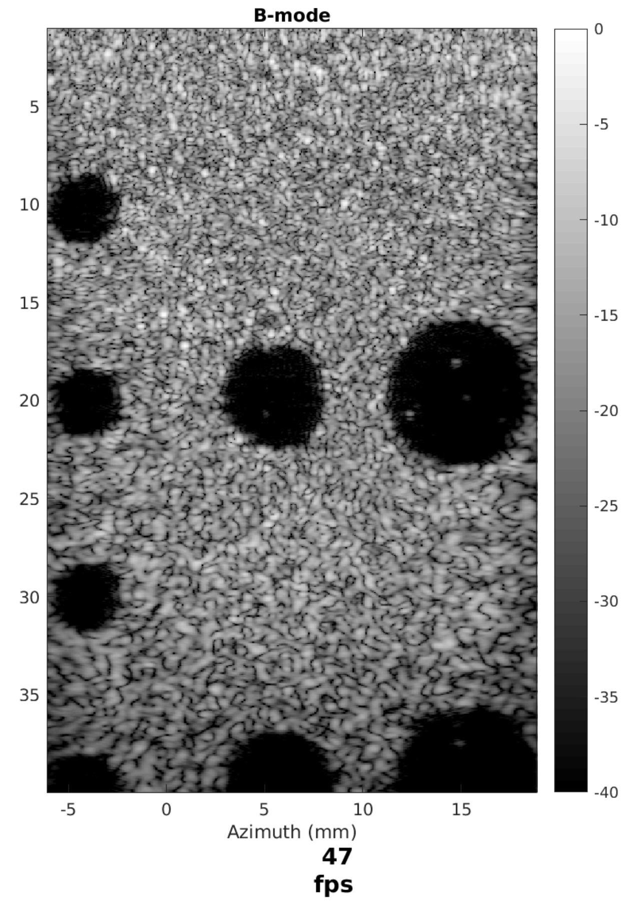

Fig. 2. An example screenshot from the real-time B-mode imaging script is displayed. Live imaging was performed at 47 frames per second for a region of 40 mm by 25 mm with 3 pixels per wavelength.

C. Other Beamforming Examples

Other beamforming configurations can be achieved by plugging in different **`DataProcessor`**s. For instance, the same B-mode image reconstruction can be instead performed using a neural network by modifying the computational graph to use the B-mode imaging network:

1) **`VSXDataFormatter<short,float2>`**
2) **`FocusSynAp<float2,float2>`**
3) **`NeuralNetwork<float2,float>`**

wherein the output of the **`NeuralNetwork`** is identical to the output of **`Bmode`**.

Similarly, the computational graph for Power Doppler imaging consists of the following **`DataProcessor`**s:

1) **`VSXDataFormatter<short,float2>`**
2) **`EnsembleFilter<float2>`**
3) **`FocusSynAp<float2,float2>`**
4) **`ChannelSum<float2,float2>`**
5) **`PowerEstimator<float2,float>`**

In this imaging case, the plane wave acquisitions are repeated multiple times to form a Doppler ensemble that is filtered in place by **`EnsembleFilter`**, focused, and converted into a Power Doppler image.

IV. Conclusion

We have presented an open source GPU-based software beamformer implementation. This work is a pedagogical demonstration of a framework that abstracts away many of the tedious low-level details commonly encountered in GPU programming, with the goal of providing a starting point for real-time algorithm deployment with GPUs.

Acknowledgment

This work is supported by the National Institute of Biomedical Imaging and Bioengineering through grant R01-EB013661. The Titan V GPU used for this research was donated by the NVIDIA Corporation.

References

[1] E. Boni, A. C. H. Yu, S. Freear, J. A. Jensen, and P. Tortoli, "Ultrasound open platforms for next-generation imaging technique development," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 65, no. 7, pp. 1078–1092, July 2018.

[2] D. Hyun, A. L. C. Crowley, M. LeFevre, J. Cleve, J. Rosenberg, and J. J. Dahl, "Improved visualization in difficult-to-image stress echocardiography patients using real-time harmonic spatial coherence imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 66, no. 3, pp. 433–441, March 2019.

[3] D. Liu and E. S. Ebbini, "Real-time 2-d temperature imaging using ultrasound," *IEEE Transactions on Biomedical Engineering*, vol. 57, no. 1, pp. 12–16, Jan 2010.

[4] 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 Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 58, no. 8, pp. 1698–1705, August 2011.

[5] 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 *2013 IEEE International Ultrasonics Symposium (IUS)*, July 2013, pp. 1280–1283.

[6] M. Walczak, M. Lewandowski, and N. Zolek, "Optimization of real-time ultrasound pcie data streaming and opencl processing for saft imaging," in *2013 IEEE International Ultrasonics Symposium (IUS)*, July 2013, pp. 2064–2067.

[7] E. Boni, L. Bassi, A. Dallai, V. Meacci, A. Ramalli, M. Scaringella, F. Guidi, S. Ricci, and P. Tortoli, "Architecture of an ultrasound system for continuous real-time high frame rate imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 64, no. 9, pp. 1276–1284, Sep. 2017.

[8] 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 Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 64, no. 1, pp. 150–163, Jan 2017.

[9] R. Göbl, N. Navab, and C. Hennersperger, "Supra: open-source software-defined ultrasound processing for real-time applications," *International Journal of Computer Assisted Radiology and Surgery*, Mar 2018. [Online]. Available: https://doi.org/10.1007/s11548-018-1750-6

[10] P. Jarosik, M. Byra, and M. Lewandowski, "Waveflow-towards integration of ultrasound processing with deep learning," in *2018 IEEE International Ultrasonics Symposium (IUS)*, Oct 2018, pp. 1–3.

[11] D. Hyun, L. L. Brickson, K. T. Looby, and J. J. Dahl, "Beamforming and speckle reduction using neural networks," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 66, no. 5, pp. 898–910, 2019.