GPU Accelerated Post-Processing for Multifrequency Biplanar Interferometric Imaging

Pingfan Meng [1], George R. Cutter Jr. [2], Ryan Kastner [1], David A. Demer [2]

[1] *Department of Computer Science and Engineering*
University of California, San Diego

[2] *Southwest Fisheries Science Center*
National Oceanic and Atmospheric Administration

***Abstract*—Multifrequency biplanar interferometric imaging (MBI) is a contemporary acoustic-data processing technology designed to remotely image the seabed or objects in the ocean. MBI requires a computationally intense data conditioning algorithm. For example, it takes a serial central processing unit (CPU) 10 seconds to process each transmission for typical survey data from of a five-beam/frequency group for split-aperture echosounders (e.g., Simrad EK60). However, because many of the computations may be processed in parallel, we utilized a graphic processing unit (GPU) to accelerate the computations. Here we present the GPU accelerated approach and discuss its optimization. The GPU implementation was 40-fold faster than the baseline serial CPU implementation.**

I. Introduction

Acoustic multifrequency biplanar inter imaging (MBI) is a contemporary technology for remote underwater object detection, classification, and mapping. MBI was developed to process data from split-beam scientific echosounders (e.g., Simrad EK60 and ME70) used in fisheries, oceanography, and seabed mapping investigations [1]. MBI solutions enable three-dimensional (3-D) mapping of surfaces, such as the seabed, or groups of targets, such as fish schools. For instance, Figure 1 displays an MBI map of a wrecked ship constructed from only a few transmissions. MBI enables single-transmission 3-D mapping and is applicable to the data from multi-frequency single- and multi-beam echosounders.

The high resolution MBI is desired for multiple oceanographic research activities (e.g., tracking fish schools relative to their seabed habitat). To achieve this high resolution, a numerous estimation method is applied to the raw sample data received from the underwater acoustic device. Although the resolution is improved, the numerous estimation algorithm is computationally intensive. For example, on a high-end serial CPU, the processing of MBI data from a single EK60 transmission takes roughly 10 seconds. Although this long processing time makes the MBI technology impractical on a serial processor, the algorithm may be accelerated on parallel hardware. Note, however that these serial processing times are expressed for the full data matrix, including samples from all beams and ranges; hence pre-filtering and processing a sparse data matrix could make MBI practical on serial system.

Graphic Processing Units (GPUs) employ a many-core architecture which highly parallelizes the computations. This architecture is particularly suitable for window-like or matrix-like data processing. Thus, GPUs have been utilized as promising solutions in many accelerated image processing applications [2], [3], [4]. In the MBI algorithm, the sonar sample data are stored and computed in a two-dimensional array. Therefore, the MBI algorithm is similar to an image processing algorithm. Moreover, a GPU is easy to program compared to other types of hardware accelerating platforms such as a field-programmable gate array (FPGA). Computer Unified Device Architecture (CUDA) is a C-like programming language that provides a development tool for software programmers to easily manipulate GPUs. Due to their suitability and the programmability, we selected a GPU to accelerate the MBI algorithm.

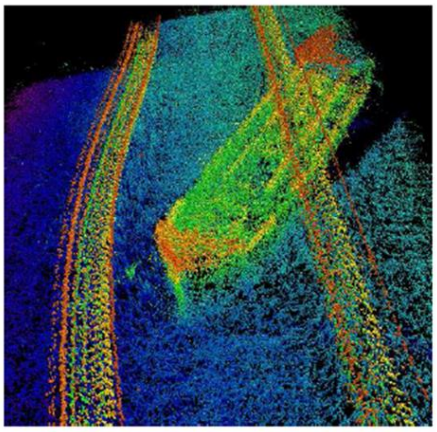

Fig. 1. MBI mapping of a ship wreck. The shallow stripes are artifacts from the transmit pulses and near-surface reverberation.

In next, we describe the MBI algorithm in section II; and introduce our GPU implementation in section III; show the performance results for the GPU implementation in section IV; and present our conclusion in section V.

II. MBI Algorithm

MBI uses power, and two orthogonal (longitudinal and transverse) interferometric phase angles data from the received echoes. The results resolve a 3-D map of the targets found

978-0-933957-40-4 ©2013 MTS

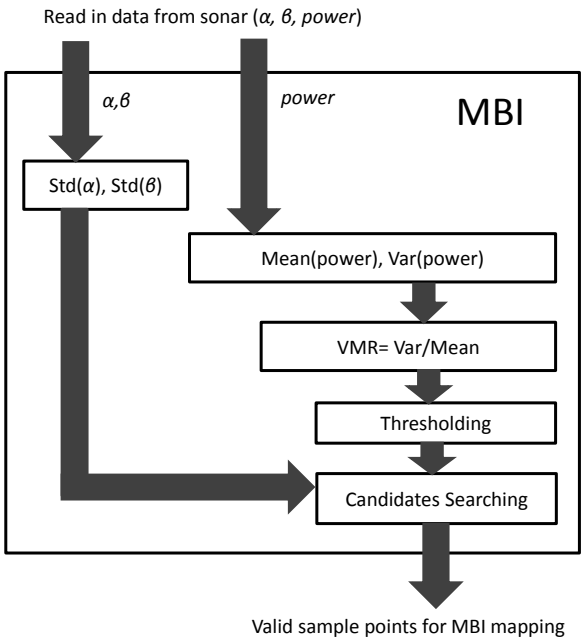

Fig. 2. MBI algorithm overview.

within the common beam space [5][6]. Statistics of the power and angle data samples are employed for filtering, estimating coherency, and classifying samples. The filtering rejects noise and unresolvable targets. This is critical for generating high quality 3-D acoustic images [7].

The MBI algorithm is described in figure 2. The input is sample data from one echosounder transmission. Each sample point consists of the interferometric angles (α, β) and power data. Firstly, the mean and variance values of the power data are computed to generate the variance mean ratio (VMR). These VMR values are filtered by a threshold. The sample points which pass the threshold are saved in a candidate pool. The standard deviation values of the angle data are computed to judge whether each candidate is valid. The valid points are fed to the MBI visual mapping module.

In the MBI algorithm, the numerous estimations (Std, $Mean$ and Var) are the core part of the computation. Therefore, we investigated their computation and memory access features to understand the complexity. The computational behavior and the memory access pattern of the numerous estimations are visualized in Figure 3. The input data is organized as a 2-D array whose dimensions correspond to the detection range and beam angle. The estimation operations are conducted on every adjacent three elements across the beam angle dimension of the data array, shown as the dash line rectangle in Figure 3. In the serial-execution model, the estimation operation (dash line rectangle) needs to iterate across the range and the beam angle dimension. Therefore, the asymptotic complexity of the algorithm is $O(A * D)$ where A is the length in the angle dimension and D is the length in the range dimension. Since the product of A and D is on the order of thousands, analytically, the estimation operations are the bottlenecks.

In addition to the asymptotic analysis, we also profiled

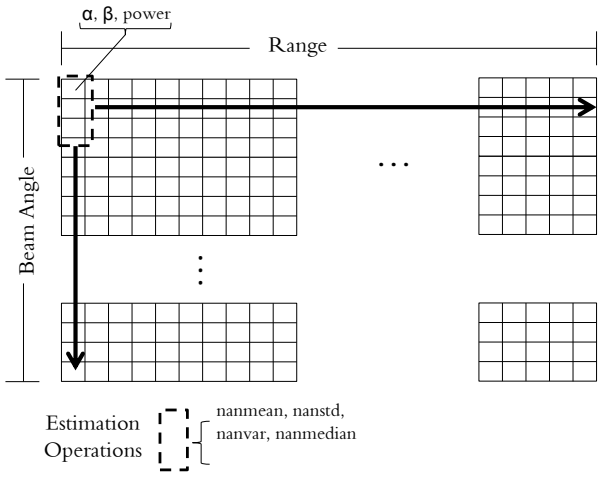

Fig. 3. Numerous estimation operations.

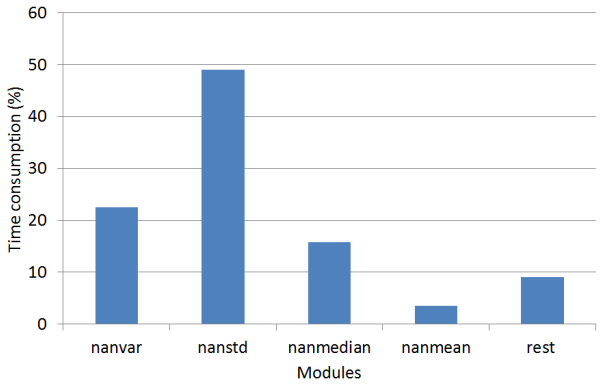

Fig. 4. MBI algorithm bottleneck study. The numerous estimation operations are identified as the bottlenecks.

the MBI algorithm on the CPU to confirm the bottlenecks experimentally. As shown in Figure 4, the parallel numerical estimation operations (nan operations) [8] consume 91% of the total run-time of the entire algorithm. The rest of the algorithm consists of non-parallel operations. These operations are not suitable for the GPU due to the fact that the thresholding causes conditional branches. Therefore, we partitioned the algorithm and kept these non-parallel operations on the CPU. We accelerated the parallel operations on the GPU.

III. GPU Implementation

In this section, we present our GPU implementation of the MBI algorithm. GPU programming requires the programmer to map the algorithm to the architecture features [9]. This requires a balanced parallel thread assignment and an efficient memory access arrangement. We describe these two aspects of our implementation.

The GPU provides threads as the parallel processing unit. These threads concurrently share the same instruction but process different sets of data. Assigning computation workload to these threads is how GPU parallel resources are used. In GPU implementation of the MBI algorithm, every 3-element estimation operation is assigned to one thread. As

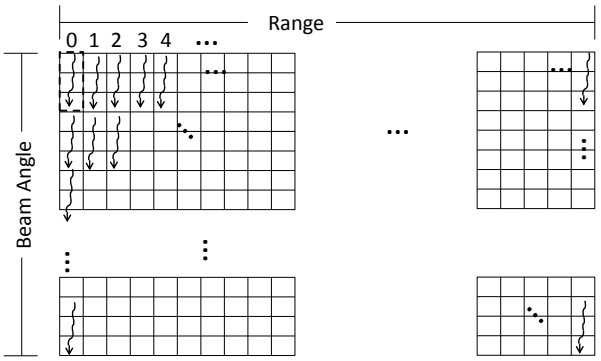

Fig. 5. GPU thread assignment. Every 3-element data group assigned on one thread. The numbers represent the thread indices which achieve memory coalescing technique.

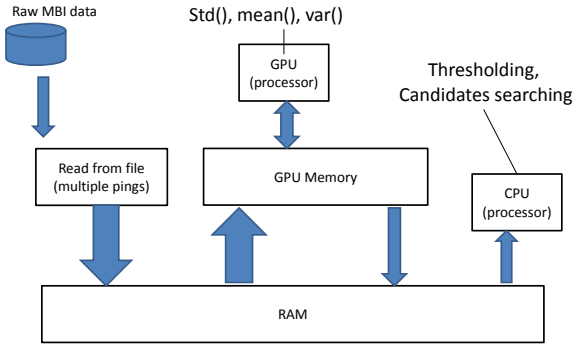

Fig. 6. GPU accelerated MBI program executional model.

demonstrated in Figure 5, we use a thread of estimation operations to compute every 3-element data group. This thread assignment allows the GPU to simultaneously process all of the estimation operations for all of the data from a transmission. Refer to Figure 3 to see that the original iteration of estimation operations is now accelerated as multiple instances of threads.

Accessing data in the GPU global memory is inevitable and costly. Thus, global memory is generally a major bottleneck for GPU implementations. To minimize this impact on the performance, we applied a memory coalescing technique. This technique accesses the memory as a set of physically adjacent entries. With this access pattern, the GPU can transfer adjacent data in one memory transaction, in parallel. In the MBI GPU implementation, we organized the data access pattern as shown in Figure 5. Adjacent threads aligned across the "range" dimension of the input data. With this setting, the adjacent threads access a chunk of consecutive memory locations, in a parallel fashion. Another memory optimization is to re-use data on the on-chip memory. In the GPU implementation, the input data and the intermediate data (e.g., the mean value for the variance computation) are stored in the registers as opposed to reading them again from the global memory.

As described in section II, the accelerated MBI algorithm consists of a GPU portion and a non-parallel CPU portion. The entire system executional model is described in Figure 6.

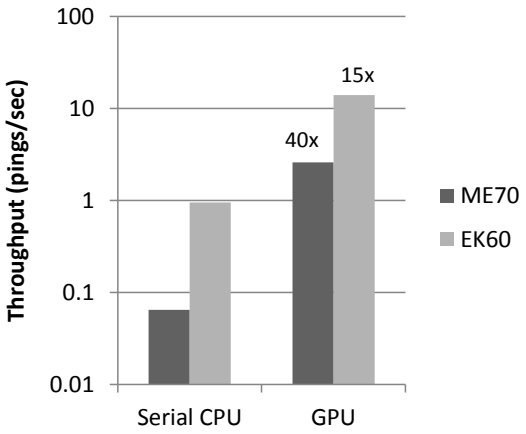

Fig. 7. GPU acceleration results.

We transfer the input data arrays from the CPU memory to the GPU memory. Then, the parallelized numerous estimation operations ($std()$, $mean()$ and $var()$) are processed on the GPU. The GPU output is transferred back to the CPU memory. We then compute the $thresholding$ and $candidates\ searching$ operations on the CPU.

IV. Results

The accelerated MBI implementation was tested on an Intel i7 quad-core 3.4GHz workstation equipped with GTX590 GPU. The CPU part of the MBI program is implemented using Matlab (Release 2012a, The MathWorks, Inc., Natick, Massachusetts, U.S.). The workstation runs Windows 7. We compiled the GPU accelerated program using $nvmex$ which provides the Matlab code an interface to call and transfer data to the GPU code.

The GPU results are compared with the equivalent CPU implementations in Figure 7. We tested ME70 mode (multi-beam) and EK60 mode (single-beam) MBI algorithms [10] on the GPU. For the ME70 mode, the GPU achieved a processing rate of 2.6 $transmissions/sec$, 40 times faster than the baseline CPU implementation. For the EK60 mode, the GPU implementation achieved 14 $transmissions/sec$, 15 times faster.

For our MBI GPU kernel, we also conducted experiments to investigate the relationship between the GPU occupancy and the performance. The GPU occupancy dominantly affects the performance. This explains how effectively the accelerating cores on the GPU are utilized. In the MBI algorithm, the occupancy is directly correlated with the size of the input data. Figure 8 shows the relationship between the GPU performance and the size of the data input to the $nanmean$ kernel. When the input is less than 10,000, the performance of the GPU is almost the same as the CPU. However, as the input size grows, the GPU performance increases rapidly. When the input size is 10 million, the GPU achieves an 11 -fold increase in speed versus the CPU. This result indicates that MBI applications

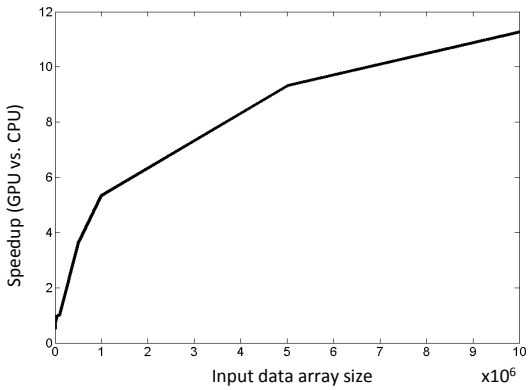

Fig. 8. Processing speed of GPU vs. CPU implementations of the MBI for different input sizes.

with large data files or high resolutions may be accelerated using our GPU implementation.

V. Conclusion

We presented a GPU-accelerated implementation of the MBI algorithm. We reported the performance of this implementation using the GTX590 GPU. We also discussed the thread assignment and memory access optimizations for our GPU development. The results indicate that the GPU implementation outperforms the equivalent CPU implementation by a factor of 40.

References

[1] G. Cutter and D. Demer, "Multifrequency biplanar interferometric imaging," *Geoscience and Remote Sensing Letters, IEEE*, vol. 7, no. 1, pp. 171–175, 2010.

[2] J. Fung and S. Mann, "Using graphics devices in reverse: Gpu-based image processing and computer vision," in *Multimedia and Expo, 2008 IEEE International Conference on*, 2008, pp. 9–12.

[3] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, ser. FPGA '12, 2012, pp. 47–56.

[4] D. Hefenbrock, J. Oberg, N. Thanh, R. Kastner, and S. Baden, "Accelerating viola-jones face detection to fpga-level using gpus," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, 2010, pp. 11–18.

[5] D. A. Demer, M. A. Soule, and R. P. Hewitt, "A multiple-frequency method for potentially improving the accuracy and precision of in situ target strength measurements," *The Journal of the Acoustical Society of America*, vol. 105, no. 2, pp. 994–994, 1999.

[6] S. G. Conti, D. A. Demer, M. A. Soule, and J. H. Conti, "An improved multiple-frequency method for measuring in situ target strengths," vol. 62, no. 8, pp. 1636–1646, 2005.

[7] D. A. Demer, G. R. Cutter, J. S. Renfree, and J. L. Butler, "A statistical-spectral method for echo classification," vol. 66, no. 6, pp. 1081–1090, 2009.

[8] MathWorks. Not a number operation @ONLINE http://www.mathworks.com/help/matlab/ref/nan.html.

[9] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08, 2008, pp. 73–82.

[10] V. M. Trenkel, V. Mazauric, and L. Berger, "The new fisheries multibeam echosounder me70: description and expected contribution to fisheries research," vol. 65, no. 4, pp. 645–655, 2008.