Real-time Implementation and Performance Optimization of 3D Sound Localization on GPUs

Yun Liang*, Zheng Cui*, Shengkui Zhao*, Kyle Rupnow*, Yihao Zhang† Douglas L. Jones‡, Deming Chen‡
*Advanced Digital Sciences Center, Illinois at Singapore
†Sun Yat-sen University
‡University of Illinois at Urbana-Champaign
*{eric.liang, cui.zheng, shengkui.zhao, k.rupnow}@adsc.com.sg, †zhyihao@sysu.edu.cn,‡{dl-jones, dchen}@illinois.edu

***Abstract*—Real-time 3D sound localization is an important technology for various applications such as camera steering systems, robotics audition, and gunshot direction. 3D sound localization adds a new dimension, but also significantly increases the computational requirements. Real-time 3D sound localization continuously processes large volumes of data for each possible 3D direction and acoustic frequency range. Such highly demanding compute requirements outpace current CPU compute abilities.**

This paper develops a real-time implementation of 3D sound localization on Graphical Processing Units (GPUs). Massively parallel GPU architectures are shown to be well suited for 3D sound localization. We optimize various aspects of GPU implementation, such as number of threads per thread block, register allocation per thread, and memory data layout for performance improvement. Experiments indicate that our GPU implementation achieves 501X and 130X speedup compared to a single-thread and a multi-thread CPU implementation respectively, thus enabling real-time operation of 3D sound localization.

I. Introduction

Real-time 3D sound localization has become important for many applications such as direction finding of human voice sources in 3D video-conferencing systems, or human-robot interactions, localization of gunshots in military or policing applications, auditory scene analysis for hearing aids and many other applications [4]. Direction information of the acoustic sources provides great utility in situation awareness and monitoring. However, the conventional real-time passive acoustic sound localization systems are mainly based on the time-difference-of-arrival (TDOA) approach and are limited to 2D. They are usually incapable for reliable sound localization in noisy environment and for the applications with 3D requirements. Hence, for better suiting to the applications it is necessary to develop robust real-time 3D sound localization systems with reliable accuracy of direction finding, easier deployment, fewer microphones and smaller system size.

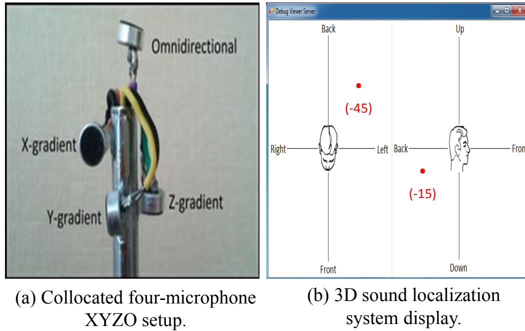
In this work, we use a collocated microphone array of four sensors which approximates the acoustic vector sensor (AVS) array that has been widely used under water [11]. Compared to the conventional linear microphone array, the AVS array is significantly more compact and smaller in size. Because the approximate AVS array exploits the relative amplitude difference between the sensors, it does not require a large aperture for high performance and is capable of simultaneous localization of multiple low frequency and high frequency wideband sound sources [7].

978-3-9810801-8-6/DATE12/©2012 EDAA

However, real-time implementation of these algorithms is very challenging due to their high computational demands. Existing high-performance 3D sound localization implementations using CPUs fall far behind the real-time requirement. The good news is 3D sound localization is highly suitable for parallel processing, as different frequencies and 3D angles are independent. Hence, we explore new opportunities of accelerating 3D sound localization using Graphics Processing Units (GPUs). Recently, GPUs have received increasing attention for scientific and data-intensive applications because they offer high parallelism with hundreds of processing cores compared to CPUs [8], [9]. We will demonstrate that real-time 3D sound localization can be achieved by our efficient implementation on GPUs. 3D sound localization also presents an excellent case study for evaluating the GPU programming techniques required and their relative importance to achieve very high efficiency for a novel application.

Although GPU architecture uses massive multithreading, optimizing GPU applications for high performance is still complex [8]. The optimal number of threads cannot be trivially determined because the overall performance depends on both thread level parallelism and single thread performance. In addition, the number of threads that are simultaneously executing on GPU is often limited by register allocation per thread. Small increase in register allocation per thread may improve the single thread performance but may cause fewer threads to be simultaneously executed as registers are a fixed shared resource. Threads may sometimes still perform memory read/write inefficiently even if multithreading is supported. Hence, we find that it is also necessary to perform memory coalescing optimization for better memory bandwidth and thus better performance. In this work, we evaluate the effect of these key architectural parameters on the GPU performance. In summary, we make the following contributions.

- We present a real-time implementation of 3D sound localization onto GPUs.
- We present GPU performance optimization by exploring the number of threads per block and register allocation per thread and reordering off-chip memory accesses.
- We build a real-time system with AVS array, CPUs and GPUs for 3D sound localization.



(a) Collocated four-microphone XYZO setup.

(b) 3D sound localization system display.

Fig. 1. Microphone setup and system display. In Figure 1 (a), the top microphone has an omni-directional response and the bottom three microphones have gradient responses. Each sensor is 6mm (Diameter) $\times$ 2.7mm (Height).

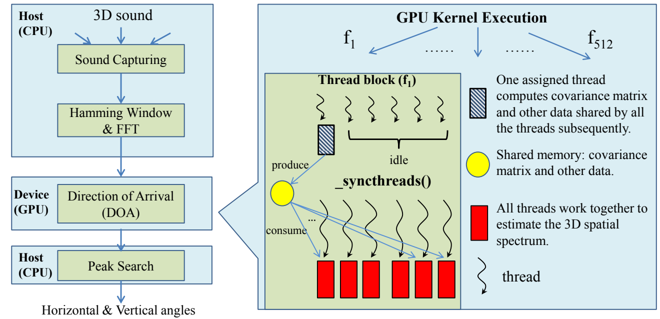


Fig. 2. GPU implementation framework. One thread block corresponds to one frequency. The threads in a thread block work together to estimate the 3D spatial spectrum.

II. 3D Sound Localization

Sound localization has been accelerated using Field Programmable Gate Array (FPGA) devices to meet its real-time requirement [4]. However, it is limited to horizontal angle only and has a front-back ambiguity. The implementation of sound localization is based on the TDOA approach. Real-time 3D sound localization has been implemented in [10] based on extended Kaman filter for two microphones. However, the localization performance tends to degrade significantly when the reverberation or noise levels increase.

Our 3D sound localization is based on a novel collocated four-sensor approximate AVS array as shown in Figure 1 (a). This compact AVS microphone sensor array consists of three orthogonally mounted acoustic particle velocity gradient microphones X, Y and Z and one omni-directional acoustic pressure microphone O. We call it the XYZO array for the rest of the paper. The main advantage of applying the gradient microphones over traditional pure pressure microphones is that they can make use of more available acoustic information including the amplitude as well as the time difference compared to only time difference as used in pure pressure microphone arrays. More importantly, the XYZO array offers better performance with a much smaller size. To apply the XYZO array for the sound localization application, an offline calibration process is performed once to obtain the steering vector of the array by measuring and interpolating the impulse response of the array in the 3D space ($[0, 180) \times [0, 360)$).

The online 3D sound localization consists of sound capturing, hamming window and FFT computation, direction of arrival (DOA) estimation, and peak search as shown in Figure 2. In the experiment, we use a commercial M-Audio sound card for sound capturing. The sound inputs received at the four microphones are first segmented by a rectangular window which has a size of 15360 signal samples. Each segment of signal data is termed as a frame. The four parallel frames from the four microphones in the XYZO array are used for a single DOA estimation. Next, each of the four frames is equally split into 30 smaller blocks of 512 samples. And then a 512-length hamming window and a 512-point fast Fourier transform (FFT) are performed on all the blocks to convert the data from time-domain into frequency-domain. This results in 512 frequency bins and 30 complex values for each frequency bin. For DOA estimation, we use the frequency-domain minimum variance distortionless response estimator [6]. For each frequency bin, the covariance matrix across the four microphones is first built. To estimate the spatial spectrum in the 3D space for each frequency bin, the covariance matrix is transformed and multiplied by the steering vector for each angle in the 3D spatial space. Then, the obtained narrowband spatial spectrums are summed across all the frequency bins to give a combined spatial spectrum. Finally, the output represented by the horizontal and vertical angles of the sound source is found by searching the peak in the combined spatial spectrum in the 3D space as shown in Figure 2. It should be noted that all the computations are complex value based.

We build a real-time system with AVS array, CPUs and GPUs for 3D sound localization. The microphone array shown in Figure 1 (a) continuously captures sounds from all directions. Then, they are processed following the steps shown in Figure 2. Finally, the estimated angle is displayed using a GUI as shown in Figure 1 (b). The left sub-figure in Figure 1 (b) displays the horizontal angle highlighted by the red dot while the right sub-figure displays the vertical angle with the head photo standing for the position of the XYZO array.

III. GPU Implementation

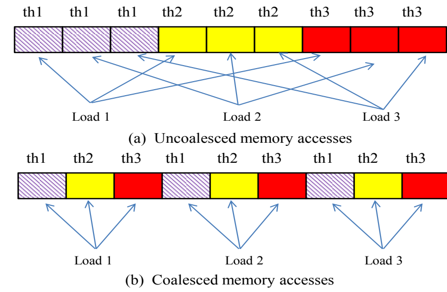
We use CUDA programming model for our GPU implementation [2]. In CUDA, threads are organized in groups called thread blocks. Each thread-block is executed on a GPU streaming multiprocessor (SM) which consists of some streaming processors (SP) that execute individual threads. Threads in a thread block share resource (e.g. registers, shared memory) together and can perform a synchronization by explicitly calling the $_syncthreads$ primitive. Within a thread block, threads are organized into warps which consist of 32 threads. Warps are the units of thread scheduling. A more detailed description of CUDA can be found in [2]. The GPU is often used as a coprocessor for accelerating kernel codes which normally contain rich data-level parallelism. During the program run, the computation load is dominated by the DOA estimation step. Thus, we only map the DOA component onto GPUs for acceleration as shown in Figure 2.

In the following, we illustrate our implementation of 3D sound localization on GPUs and the corresponding architecture related design space exploration and optimization.

Shared memory & thread blocks. DOA is highly suitable for parallel processing as the processing of different frequency bins is independent. For each frequency bin, its covariance matrix and other intermediate data need to be computed first and these data are shared by all the 3D angles for the subsequent 3D spatial spectrum estimation. Hence, we parallelize different frequency bins as different thread blocks. There are 512 thread blocks in total. Then, we store the covariance matrix and intermediate data into low-latency shared memory, which can be reused by different threads in the same thread block. More clearly, for each frequency we assign one thread to compute the covariance matrix and other intermediate data. This involves a series of matrix operations. Once the thread finishes the computation, it carries out synchronization so that other threads in the same block can share the data as shown in Figure 2. Then, the threads in the same thread block work together to estimate the 3D spatial spectrum. The required shared memory size per frequency is a constant and does not change with the number of threads per thread block.

Threads. We consider the number of threads per thread block as a parameter. Let us define it as N. For each frequency, we have to perform the 3D spatial spectrum estimation for all the angles in the 3D space ($[0, 180) \times [0, 360)$). The spectrum estimation for different angles is independent. Thus, we can parallelize different angles using different threads. For each angle, the assigned thread loads the corresponding calibration data and performs a series of matrix operations. Within a thread block, each thread is responsible for estimating the spatial spectrum for $\lceil \frac{360 \times 180}{N} \rceil$ angles in the 3D search space. The general philosophy of using a GPU for acceleration is to run as many threads as possible to hide the long memory latency. However, choosing the optimal number of threads per block (N) is not a trivial task — larger number of threads per thread block does not ensure good results as not all the threads can be active at the same time [3]. In addition, N is often limited by the architectural limits (e.g. resource, total threads) of the GPU platforms [3]. We explore every possible choice for N. The scheduling unit for threads are warps (32 threads). We always set N to be multiples of 32 to efficiently utilize the computing power.

Register. The GPU compiler (nvcc) may perform some optimizations to improve the single thread performance at the expense of extra registers. Furthermore, the nvcc compiler may perform aggressive register allocation to remove the dependencies among instructions. However, an incremental increase in the register allocation per thread may cause fewer thread blocks and thus fewer threads to be simultaneously executed on one SM as the registers are a fixed resource shared by all the threads on the same SM. Thus, the programmer has to choose a good register value, which allows the compiler to improve the single thread performance and maintain a large number of parallel threads in the same time. The nvcc compiler provides an interface to specify the maximum number of



(a) Uncoalesced memory accesses

(b) Coalesced memory accesses

Fig. 3. Memory coalescing optimization. Note that this example shows three threads for demonstration purpose.

registers per thread used for the kernel at compile-time. We explore this number together with the number of threads per thread block (N) in our implementation.

Memory Coalescing. Each angle's corresponding calibration data (8 data words) must initially be loaded before computation. Different angles use different calibration data. Thus, the calibration data are stored in off-chip global memory as there is no reuse among different threads. In our CPU implementation, the calibration data are organized thread by thread continuously as shown in Figure 3(a). This layout is good for the CPU memory hierarchy due to the temporal and spatial locality [5]. However, this layout turns out to be a bad layout for the GPU architecture. Recall that threads are grouped into warps (32 threads) and all the threads in a warp are scheduled together. If we continue with the layout shown in Figure 3 (a), threads in a warp (thread 1, 2, and 3 in Figure 3) will not access adjacent data points, generating uncoalesced memory accesses. As a result, the memory accesses of the threads in a warp have to be separated into multiple memory transactions. Uncoalesced memory accesses will seriously degrade the performance [2]. Therefore, we reorganize the data as shown in Figure 3 (b). Now, the threads in a warp access adjacent data points in the memory, and can be accomplished by a single memory transaction.

IV. Experimental Results

We evaluate two implementations: CPU and GPU. We implement both a single-thread and a multi-thread CPU version (e.g. parallelizing DOA estimation on 4 cores) using OpenMP [1]. The execution is measured on an Intel quad-core i5-750 2.67 GHz CPU with 3GB of RAM. The GPU experiments are done using the GTX480. For both the CPU and GPU implementations, we access the clock counter to obtain the precise execution time. The window size per frame is 15360 and the sampling rate is 44100/sec. Thus, one frame has to be processed within 348 ms (15360/44100) under the real-time constraint. The input audio file contains 132 frames. We observe that there is little variation in processing time across frames for both the CPU and GPU implementations. In the following, the time we report is the average processing time of one frame across a total of 132 frames.

In the following, we will first describe the performance improvement achieved by our design space exploration. We then present the CPU vs. GPU performance comparison data.

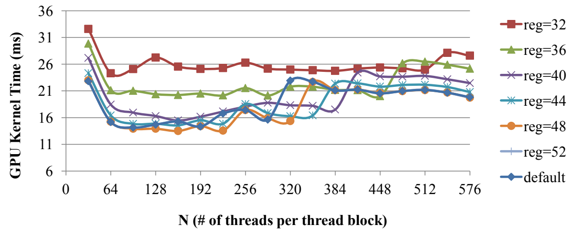


Fig. 4. GPU kernel time variation across the design space.

Threads & Register We consider threads per thread block (N) and register allocation per thread (reg) as input variables. The default register use (without register limit) of our 3D sound localization kernel is 53. There should be at least one thread block per SM during the kernel execution. Thus, the maximum value for N is 576 due to the constraint of register use [3]. Recall that N is a multiple of warp size (32). Thus, $N = 32k, 1 \leq k \leq 18$. The way that the nvcc compiler performs register allocation is unknown to the programmer, but the nvcc compiler provides a compilation option *-maxrregcount* to specify the maximum number of registers used per thread. In theory, the actual register use might be less than the limit. For all the experiments, we use nvcc compilation interface to verify the actual register use.

To further improve performance, we explore various values of N and reg. The results are shown in Figure 4. Only a subset of results is displayed. As shown, the design space exhibits high variation in terms of GPU kernel time. The overall performance critically depends on both N and reg. Moreover, the optimal setting ($N = 160$, $reg = 48$) is about 3X faster than the worst setting ($N = 32$, $reg = 32$). The designers have to perform a detailed design space exploration to find the optimal setting.

Memory Coalescing. The comparison between coalesced and uncoalesced memory accesses is shown in Figure 5. Overall, the coalesced version achieves about 10X speedup over the uncoalesced version. As shown, threads may still perform global memory operations inefficiently when the global memory latency cannot be perfectly hidden by multithreading. For our 3D sound localization, we find that it is crucial to transform the data layout for better memory bandwidth.

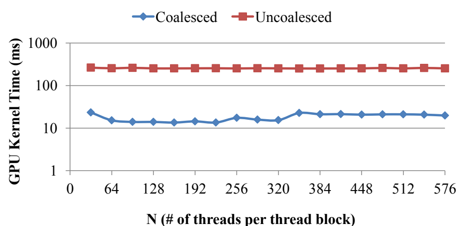


Fig. 5. Memory coalescing vs uncoalescing.

Speedup. The runtime comparison between CPU and GPU for both kernel and entire application is shown in Table I. For the GPU implementation, we choose $N = 160$, $reg = 48$, and perform the memory coalescing transformation. As shown, the single-thread CPU implementation is very slow, which falls far below the real-time requirement (21.2X slower). The multi-thread CPU implementation using OpenMP is 5.5X slower than real-time requirement. More importantly, further data processing workload (e.g. sound enhancement, 3D sound construction, and multi-source sound localization) are promising research directions currently. With such workload, the CPU implementation will be even slower.

	CPU (ms)		GPU (ms)	Speedup	
	Single	Multi		Single	Multi
Kernel	7366	1918	13.56	543	141
Application	7367	1919	14.71	501	130

TABLE I
RUNTIME COMPARISON. REAL-TIME REQUIREMENT IS 348 MS.

Overall, our GPU implementation achieves significant speedup for both the kernel and the entire application. Application wise, the GPU version achieves a 501X and 130X speedup compared to the single-thread and multi-thread CPU implementation, respectively, and successfully meets the real-time processing requirement. Compared to the CPU implementation, the GPU implementation incurs 4% additional overhead for transferring data between the host and the device. Such overhead is included in the GPU runtime.

V. CONCLUSION

3D sound localization is an important technique, which finds applications in various areas. However, its highly demanding computational requirement outpaces current CPU compute abilities. In this paper, we proposed a real-time implementation of 3D sound localization using GPUs. Through our experiments, we demonstrated that GPU platforms were well suited for 3D sound localization due to their massively parallel architectures. We also performed a GPU performance optimization by exploring the number of threads per thread block and register allocation per thread, and reordering data layout. Experiments indicated that significant speedup compared to CPU implementation was achieved.

VI. ACKNOWLEDGMENTS

The Advanced Digital Sciences Center is funded by A*STAR under the Human Sixth Sense Project.

REFERENCES

[1] http://openmp.org/wp/.
[2] NVIDIA. NVIDIA CUDA Programming Guide, Version 3.2.
[3] NVIDIA. Occupancy Calculator. http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html.
[4] P. Aarabi D. Nguyen and A. Sheikholeslami. Real-time sound localization using field-programmable gate arrays. In *ICASSP*, 2003.
[5] Y. Liang and T. Mitra. Improved procedure placement for set associative caches. In *CASES*, 2010.
[6] M. Lockwood and D.L. Jones. Beamformer performance with acoustic vector sensors in air. In *Journal of Acoustic Society of Americ*, 2006.
[7] S. Mohan et al. Localization of multiple acoustic sources with small array using a coherence test. In *Journal of Acoustic Society of Americ*, 2008.
[8] S. Ryoo et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, 2008.
[9] S. Ryoo et al. Program optimization space pruning for a multithreaded GPU. In *CGO*, 2008.
[10] M. Usman. Real time 3D humanoid sound source localization and tracking in actual environments. In *IEEE Canadian Conference on Electrical and Computer Engineering*, 2008.
[11] K. T. Wong and M. D. Zoltowski. Self-initiating music-based direction finding in underwater acoustic particle velocity-field beam-space. In *IEEE Journal of Oceanic Engineering*, 2000.