GPU-based Acceleration of PyHARK

*Zirui LIN[1], Katsutoshi ITOYAMA[1,2], Kazuhiro NAKADAI[1], Masayuki TAKIGAHIRA[2], Haris GULZAR[3], Takeharu EDA[3], Monikka Roslianna BUSTO[3], Hideharu AMANO[4]

1 Tokyo Institute of Technology, 2 Honda Research Institute Japan,
3 Nippon Telegraph and Telephone Corporation, 4 Keio University.

1. Introduction

Since its release in 2008, the open-source robot audition software HARK [1, 2] has been continuously developed. HARK provides a programmable platform for users to build robot audition systems on demand. PyHARK, as a new feature of HARK version 3.4 [3], provides a Python interface for HARK. It allows users to use HARK functions through Python programs and utilize commonly used development tools such as Jupyter Notebook and Visual Studio Code to enhance programming efficiency.

This paper introduces the GPU-based acceleration of PyHARK, which aims to reduce the processing time of PyHARK further. We deployed GPU-based acceleration on functions involving matrix operations of sound source localization (SSL) and sound source separation (SSS) modules of PyHARK, the bottle of PyHARK processing. We elaborate on our implementation methods and provide experiment results. In the results, for 12.5 second, 60-channel audio data, NVIDIA A100 GPU in a server configured with AMD EPYC 7352 CPU achieved 1.6× and 1.2× acceleration for SSL and SSS, while Jetson AGX Xavier is 2.0× and 1.1×.

2. Related Work

Research in many fields has attempted to use GPUs to accelerate data processing. This section provides an overview of related work, focusing on efforts to improve the processing time of audio signal processing, including HARK.

The acceleration of HARK has been studied. Hou et al. [4] proposed a method to deploy the HARK module of SSL on FPGA. They partially realized the acceleration of sound source localization. Similarly, Qin et al. [5] proposed a method to deploy the SSS module in HARK on an FPGA, achieving high performance and low power consumption computation. However, the computing resources of FPGA are relatively limited. According to [5], due to the limitation of FPGA resources, they almost used the entire FPGA to implement four sound source separation cores. DSP resources became the bottleneck, making it difficult to achieve large-scale processing. However, GPU usually has a more significant number of processing units and memory capacity, enabling it to attain larger-scale processing. At the same time, GPU is more common in general-purpose computers than FPGA. Therefore, acceleration using GPU has the potential to benefit a broader user base than using FPGA.

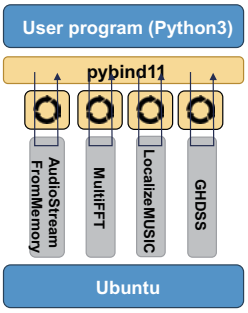


Fig. 1: Processing Flow of PyHARK

Some studies have used GPU to improve processing efficiency in audio signal processing. For example, Huang et al. [6] proposed an optimized real-time MUSIC (MUltiple SIgnal Classification) algorithm for SSL that uses a CPU-GPU architecture for processing. Their algorithm improvement adapts to the characteristics of CPU and GPU and dramatically improves the algorithm's performance through collaborative computing. However, they only verified their method on the 4-channel audio signal. When more microphones are used, they need to optimize it separately, such as a 60-channel microphone array.

Raj et al. [7] proposed a GPU-accelerated guided SSS for meeting transcription. This implementation was nearly 300 times faster than the original implementation on the CHiME-6 benchmark, eliminating the computational bottleneck of this technique. However, they use GSS(Guided Source Separation) algorithm for SSS, which is different from our target algorithm, GHDSS (Geometric High-order Dicorrelation-based Source Separation), and thus other considerations will be necessary.

Our work is the first dedicated GPU-based acceleration of PyHARK and simultaneously supports SSL and SSS modules of PyHARK.

3. Architecture of PyHARK

3·1 Processing Flow of PyHARK

Fig. 1 illustrates the structure of the offline batch processing version PyHARK. It has four layers: user program, pybind, HARK module, and operating system. The user program layer is a programming platform for users. They can write Python programs in this layer to use the functions of PyHARK to realize audio data processing. The pybind layer enables the

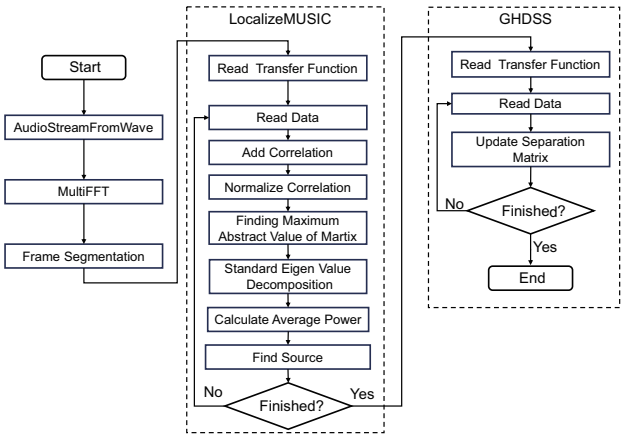


Fig. 2: Example of PyHARK Program

user program layer to call the modules of PyHARK as nodes whose source code is designed at the C/C++ level. Loops between pybind and nodes mean that for every node, offline batch processing version PyHARK does frame-by-frame processing automatically, and this is controlled in pibind layer. The HARK modules layer contains various audio processing modules, including SSL and SSS, implemented in C/C++. Users can use these modules flexibly according to the needs of the audio processing task. The operating system layer is the base of the PyHARK system. It provides the runtime environment for the above three layers and handles system-level tasks such as memory management, process scheduling, and hardware interaction.

3·2 Example of PyHARK Program

Fig. 2 shows an example of a PyHARK program that performs SSL and SSS. For SSL, we used SEVD-based (Standard Eigen Value Decomposition) MUSIC algorithm, which has been implemented in the LocalizeMUSIC node. For SSS, we used GHDSS algorithm , which has been implemented in the GHDSS node. The PyHARK program starts with reading waveform data and then segmenting data into frames. Next, the program uses a MultiFFT node to convert the time domain signal into a frequency domain signal. The output of this node is an input of LocalizeMUSIC and GHDSS nodes. Next, it comes to the LocalzeMUSIC node. First, the node reads the transfer function. Then it calculates normalized correlation matrices from the input audio data. After that, the node localizes sound sources by performing SEVD for the correlation matrix, calculating the MUSIC spatial spectrum from eigenvalues, eigenvectors, and steering vectors, and finding peaks in the MUSIC spectrum to estimate sound source directions. At last, the estimated sound source directions and the audio data are sent to the GHDSS node for separation. The GHDSS node uses the results of the LocalizeMUSIC node and the input audio signal to estimate a separation matrix. The node updates a separation matrix, which can be applied to the input audio signal to store the separated signal in wav files. At last, if the processing of all data has been finished, the program will end.

Table 1: Processing Time of Nodes and Functions of PyHARK on Jetson AGX Xavier for 60-Channel, 12.5s Audio Data Using CPU

Level	Node Name	Time (s)	Percentage
Node	**AudioStreamFromWave**	**0.118**	**0.036%**
Node	**MultiFFT**	**2.682**	**0.83%**
Node	**LocalizeMUSIC**	**241.562**	**74.70%**
	Sub Function of LocalizeMUSIC		
Sub Function	Read Transfer Function	86.396	26.72%
Sub Function	Add Correaltion	2.233	0.69%
Sub Function	Normalize Correlation	0.046	0.014%
Sub Function	Max of Abs Value	0.130	0.04%
Sub Function	Standard Eigen Value Decomposition	14.418	4.46%
Sub Function	Calculate Average Power	119.813	37.05%
Sub Function	others	18.565	5.74%
Node	**GHDSS**	**78.947**	**24.41%**
	Sub Function of GHDSS		
Sub Function	Initialize Transfer Function (GHDSS)	60.709	18.77%
Sub Function	Read Data	4.919	1.52%
Sub Function	Update Separation Matrix	9.721	3.00%
Sub Function	others	3.598	1.11%
Program		323.385	100%

4. The Bottleneck of PyHARK Processing

Table 1 shows the PyHARK program performance test result when processing 60-channel, 12.5s audio data on Jetson AGX Xavier using CPU. In this table, we can see that The execution time of node LocalizeMUSIC and GHDSS accounted for 74.70% and 24.41% of the total time, respectively. In the entire program, functions involving matrix operations take much time. They are functions of Add Correlation, Normalize Correlation, Finding Maximum Abstract Value of Matrices, Standard Eigen Value Decomposition, Calculate Average Power, and Update Separation Matrix. The total running time of these functions accounts for 45.25% of the entire program, while the total running time of parts of the reading transfer function is 45.49%. However, reading transfer function happens only when the program is executed, not every time frame, and this can be ignored even if it takes time. Therefore, we can think of processing involving matrix operations as the performance bottleneck of PyHARK. Optimizing this bottleneck is the focus of this paper.

5. Proposed Method

To optimize the bottleneck caused by matrix operations, we propose deploying them on NVIDIA GPU using CUDA to reduce processing time.

5·1 Design of Parallel Computing

For all functions with matrix operations in LocalizeMUSIC and GHDSS nodes, we can divide them into four types according to parallelism, design parallel computing for the four types separately, and allo-

Table 2: Dimensions of Input and Output Matrices of Functions and Allocation of Threads Blocks for Functions: *channels* indicates the number of microphones, *pslength* indicates sampling points of frequency, *win_size* indicates the size of the frequency range used in localization, *num_src* indicates the size of the number of sources found in localization, *height_num*, *direction_num*, and *range_num* indicate the coordinates of the sound source on the three-dimensional coordinate axis.

Function	Dimension of Input Matrix	Dimension of Output Matrix	Block Size (block.x, block.y)	Number of Blocks
Add Correlation	$pslength \times channels \times win\_size$	$calc\_freq \times channels \times channels$	(32, 4)	$\left(\frac{channels-1}{block.x}+1\right) \times \left(\frac{channels-1}{block.y}+1\right) \times calc\_freq$
Normalize Correlation	$calc\_freq \times channels \times channels$	$calc\_freq \times channels \times channels$	(32, 4)	$\left(\frac{channels-1}{block.x}+1\right) \times \left(\frac{channels-1}{block.y}+1\right) \times calc\_freq$
Max of Abs Value	$calc\_freq \times channels \times channels$	$calc\_freq \times channels \times channels$	(32, 4)	$calc\_freq$
SEVD1	$calc\_freq \times channels \times channels$	$calc\_freq \times channels \times channels$, $calc\_freq \times channels$	(32, 4)	$\left(\frac{channels-1}{block.x}+1\right) \times \left(\frac{channels-1}{block.y}+1\right) \times calc\_freq$
SEVD2	$calc\_freq \times channels \times channels$, $calc\_freq \times channels$	$calc\_freq \times channels \times channels$, $calc\_freq \times channels$	(64, 1)	$channels \times calc\_freq$
Calculate Average Power	$calc\_freq \times channels \times channels$, $calc\_freq \times channels$	$height\_num \times direction\_num \times range\_num$	(32, 4)	$height\_num \times direction\_num \times range\_num$
Update Separation Martix	$channels \times pslength$	$num\_src \times pslength \times channels$	(32,1)	$\frac{pslength+block.x-1}{block.x}$

cate thread blocks with different strategies.

Parallelism across frequency bins × channels × channels. Each element in the matrix (channels × channels) of each frequency bin is computed independently.

- Corresponding functions: Add Correlation, Normalize Correlation, Max of Abs Value, SEVD1 (generating identity matrix).

Parallelism across frequency and channels. There is only one dimension of parallelism in the matrix of each frequency bin. It means that in each frequency bin's matrix, each channel's rows can be computed independently.

- Corresponding functions: SEVD2 (eigenvalue calculation, eigenvector calculation and sorting).

Parallelism only across frequency bin. Only the matrix of each frequency bin can be computed independently.

- Corresponding functions: Update Separation Matrix GHDSS.

Parallelism across height × direction × range × channels × channels .

Rows for each channel of the matrix on the three-dimensional coordinate points can be computed independently.

- Corresponding functions: Calculate Average Power.

5·2 Implementation of Parallel Computing

For the first type, the computation of each frequency bin is allocated to thread blocks, where each thread within the block corresponds to the calculation of each element in the matrix. For the second type, the computation of each frequency bin is allocated to blocks, where each thread within the block corresponds to the computation of each row corresponding to the channel. For the third type, the computation of each frequency bin is allocated to a single thread. For the fourth type, the computation of a point on the three-dimensional coordinate axis is assigned to blocks, where each thread within the block corresponds to the computation of each element in the matrix. In this situation, if the number of parallelizable tasks in a thread block exceeds the number of threads, each thread will process multiple tasks.

Table 2 shows the dimensions of the input and output matrices for each function and the allocation of thread blocks. The number of blocks is designed to try to cover all parallel units.

6. Evaluation Experiments

To validate our proposed method, we conduct experiments to measure the processing time of nodes of LocalizeMUSIC and GHDSS on two CPUs, NVIDIA A100, and Jetson AGX Xavier.

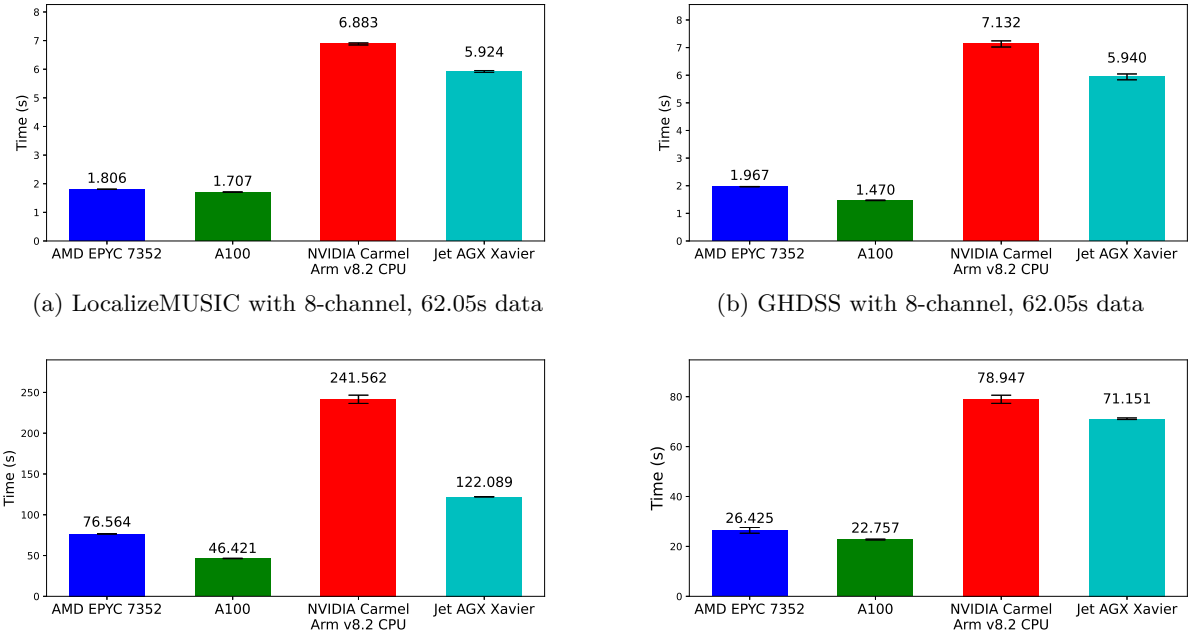
6·1 Experimental Condition

We conduct experiments on two distinct types of devices: a server configured with an NVIDIA A100 GPU and AMD EPYC 7352 CPU, and Jetson AGX Xavier, an embedded GPU device equipped with 8-core NVIDIA Carmel ARM v8.2 CPU. The NVIDIA A100 GPU, as a powerful server-grade GPU, represents high-performance computing environments. Conversely, as an embedded device, the Jetson AGX Xavier represents edge computing scenarios where resources are more limited. Thus, by evaluating our proposed method on these two devices, we try to prove that GPU-accelerated PyHARK is adaptable to various scenarios, such as cloud computing, edge computing and IoT devices.

In experiments, we measure the processing time of the LocalizeMUSIC and GHDSS nodes with and without GPU acceleration to evaluate the effectiveness of GPU-based acceleration of PyHARK. We conduct experiments on each device, processing 8-channel, 62.05 seconds audio data, and 60-channel, 12.5 seconds data respectively. Experiments were conducted 10 times under each condition.

6·2 Experimental Results

Fig. 3 shows that in all cases, the processing time of the nodes with GPU acceleration is less than without GPU acceleration. For 8-channel, 62.5s data, A100 achieved 1.1× and 1.3× acceleration for SSL and SSS,



(a) LocalizeMUSIC with 8-channel, 62.05s data

(b) GHDSS with 8-channel, 62.05s data

(c) LocalizeMUSIC with 60-channel, 12.5s data

(d) GHDSS with 60-channel, 12.5s data

Fig. 3: Results of Evaluation Experiments

while Jetson AGX Xavier is 1.2× and 1.2×. For 60-channel, 12.5s data, A100 achieved 1.6× and 1.2× acceleration for SSL and SSS, while Jetson AGX Xavier is 2.0× and 1.1×.

Therefore we can consider the GPU-based acceleration of PyHARK successfully reduced the processing time of SSL and SSS for the processing of 8-channel and 60-channel audio data on NVIDIA A100 server and Jetson AGX Xavier.

7. Conclusion

This paper presents GPU-based acceleration of PyHARK to reduce the processing time further. Evaluation experiments show that GPU-based acceleration successfully reduces the processing time of nodes of SSL and SSS of PyHARK, thereby reduces the overall running time of the PyHARK program. Experiment results also demonstrate that GPU-based acceleration of PyHARK is effective on these two devices of different scales, making GPU-accelerated PyHARK adaptable to different application scenarios.

Acknowledgement

This work was supported by JST CREST JPMJCR19K1.

References

[1] K. Nakadai, T. Takahashi, H. G. Okuno, H. Nakajima, Y. Hasegawa, and H. Tsujino. Design and implementation of robot audition system 'hark'. *Advanced Robotics*, 24:739–761, 2010.

[2] 中臺一博 and 奥乃博. ロボット聴覚用オープンソースソフトウエア hark の展開. デジタルフラクティス, 2(2):133–140, 2011.

[3] 中臺一博, 糸山克寿, and 瀧ヶ平将行. Pyhark: Hark のオンライン・オフライン処理用 python パッケージ. 人工知能学会第二種研究会資料, 2022(Challenge-061):04, 2022.

[4] Zhongyang Hou, Kaijie Wei, Hideharu Amano, and Kazuhiro Nakadai. An fpga off-loading of hark sound source localization. In *2022 Tenth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 236–240. IEEE, 2022.

[5] Ziquan Qin, Kaijie Wei, Hideharu Amano, and Kazuhiro Nakadai. Low power implementation of geometric high-order decorrelation-based source separation on an fpga board. In *2023 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, pages 1–6. IEEE, 2023.

[6] Qinghua Huang and Naida Lu. Optimized real-time music algorithm with cpu-gpu architecture. *IEEE Access*, 9:54067–54077, 2021.

[7] Desh Raj, Daniel Povey, and Sanjeev Khudanpur. Gpu-accelerated guided source separation for meeting transcription. *arXiv preprint arXiv:2212.05271*, 2022.