PyHARK Acceleration: A GPU-based Approach

Lin Zirui[1,a)] Itoyama Katsutoshi[1,2] Nakadai Kazuhiro[1] Takigahira Masayuki[2]
Gulzar Haris[3] Eda Takeharu[3] Busto Monikka Roslianna[3] Amano Hideharu[4]

Abstract: This paper presents a GPU-based approach for PyHARK acceleration, which offers a Python interface for HARK. One issue for PyHARK is that it needs a shorter processing time to achieve real-time for large-scale processing. In this paper, we designed and implemented GPU-accelerated PyHARK that significantly reduced the processing time.

Keywords: Robot Audition, Sound Source Localization, Sound Source Separation, GPU, Acceleration

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, which allows using HARK functions through Python programs and utilizing commonly used development tools to enhance programming efficiency.

This paper introduces a GPU-based approach for PyHARK acceleration. We deployed bottleneck functions of PyHARK on GPU. We elaborate on our methods and provide experiment results. In the results, for 60-channel audio data, NVIDIA A100 GPU in a server configured with AMD EPYC 7352 CPU achieved 3.6× acceleration for SSL and SSS, while Jetson AGX Xavier is 5.0×.

2. Related Work

The acceleration of HARK has been studied. Hou et al. [4] proposed a method to deploy the Sound Source Localization (SSL) module of HARK on FPGA. They partially realized the acceleration of sound source localization. Similarly, Qin et al. [5] proposed a method to deploy the Sound Source Separation (SSS) module of HARK on FPGA, achieving high performance and low power consumption computation. However, the computing resources of FPGA are relatively limited, and only 8-channel microphone arrays are supported. According to [5], due to the limitation of FPGA resources, they almost used the entire FPGA to implement four SSS cores. DSP resources become 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 acceleration for large-scale processing.

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	I/O		86.396	26.72%	
Sub Function	Matrix Operations	AddCorrealtion	2.233	0.69%	42.254%
Sub Function		NormalizeCorrelation	0.046	0.014%	
Sub Function		MaxOfAbsValue	0.130	0.04%	
Sub Function		Evd	14.418	4.46%	
Sub Function		CalcAveragePower	119.813	37.05%	
Sub Function	Non-Martix Operations		18.565	5.74%	
Node	GHDSS		78.947	24.41%	
	Sub Function of GHDSS				
Sub Function	I/O		60.709	18.77%	
Sub Function	Matrix Operations	dUpdateSeparation MatGHDSS	9.721	3.00%	
Sub Function	Non-Martix Operations		8.517	2.63%	
Program			323.385	100%	

3. The Bottleneck of PyHARK Processing

Table 1 shows the PyHARK program performance test result when processing large-scale audio data with 60 channels on Jetson AGX Xavier with CPU. In the entire program, functions involving matrix operations take much time. They are functions of AddCorrelation, NormalizeCorrelation, MaxOfAbsValue, Evd, CalcAveragePower, and dUpdateSeparationMatGHDSS. The total running time of these functions accounts for 45.25% of the entire program, while the total time of I/O accounts for 45.49%. However, I/O happens only when the program starts, not every time frame, so its time is unrelated to data length. Therefore, we can think of processing involving matrix operations as the performance bottleneck of PyHARK. Optimizing this bottleneck is the focus of this paper.

4. Proposed Method

To optimize the bottleneck caused by matrix operation functions, we deploy them on NVIDIA GPU using CUDA to accelerate the processing while supporting up to 60 microphones. Table 2 shows the matrix dimension in the processing, the unrolling strategy of each function and the size and allocation of thread blocks.

[1] Tokyo Institute of Technology, Meguro, Tokyo 152–8550, Japan
[2] Honda Research Institute Japan, Wakō, Saitama 351-0188, Japan
[3] Nippon Telegraph and Telephone Corporation, Chiyoda, Tokyo 100–8116, Japan
[4] Keio University, Minato, Tokyo 108–0073, Japan
[a)] linzirui@ra.sc.e.titech.ac.jp

Table 2 Dimensions of Input and Output Matrices of Functions and Allocation of Threads Blocks: *frequency bins* indicates the frequency range size used for SSL, *channels* indicates the number of microphones and *height_num* indicate the number of directions that the microphone array can detect. In the column of Matrix Dimension in Processing, parts inside brackets mean these parts are unrolled, and parts outside brackets mean they are the minimum parallel unit.

Function	Matrix Demension in Processing	Block Size (block.x, block.y)	Number of Blocks
AddCorrelation	(frequency_bins × channels × channels) × 1	(32, 4)	((channels-1)/(block.x)+1) × ((channels-1)/(block.y)+1) × frequency bins
NormalizeCorrelation	(frequency bins × channels × channels) × 1	(32, 4)	((channels-1)/(block.x)+1) × ((channels-1)/(block.y)+1) × frequency bins
MaxOfAbsValue	(frequency bins × channels × channels) × 1	(32, 4)	((channels-1)/(block.x)+1) × ((channels-1)/(block.y)+1) × frequency bins
Evd	(frequency bins × channels) × channels	(32, 1)	((channels-1)/(block.x)+1) × channels × frequency bins
CalcAveragePower	(frequency bin × channels× channels) × height_num	(32, 4)	((channels-1)/(block.x)+1) × ((channels-1)/(block.y)+1) × frequency bins
dUpdateSeparationMatGHDSS	(frequency bins) × channels × channels	(32, 1)	(frequency bins + block.x - 1)/(block.x)

4.1 Unrolling Strategy

We design unrolling strategies for the functions with matrix operations separately according to their independence of matrix operations and allocate thread blocks to cover all parallel units. Independence means that when performing matrix operations, the calculation of certain elements, rows/columns or subsets of a matrix does not depend on the calculation results of other elements, rows/columns or subsets.

4.2 Block Allocation

When the number of microphones increases, the matrix dimension in processing becomes larger. Although allocating a single thread block is the simplest way that can also reduce synchronization overhead, it's infeasible for scenarios with more than 1024 parallel units in the processing. This is due to the GPU's limit of 1024 threads per block. To perform large-scale matrix processing without exceeding the limit, we used many small thread blocks to cover the large matrix. Table 2 shows the size and allocation of blocks. *block.x* is fixed to 32, which is the number of threads of a warp in GPU. If *block.y* is considered in block allocation equations in Table 2, it is set to a factor (here is 4) of microphone number (60) while keeping the thread count under 1024. Otherwise, it is set to 1.

5. Evaluation Experiments

To validate our proposed approach, we conduct experiments to measure SSL and SSS processing time on two CPUs, NVIDIA A100 and Jetson AGX Xavier.

5.1 Experimental Settings

We conduct experiments on two distinct types of devices: a server configured with an NVIDIA A100 GPU and AMD EPYC 7352 CPU, which represents high-performance computing environments, and Jetson AGX Xavier, an embedded GPU device equipped with 8-core NVIDIA Carmel ARM v8.2 CPU, represents edge computing scenarios where resources are more limited.

In experiments, we measure processing time with and without GPU acceleration. We conduct experiments on each device with 60-channel 12.5 seconds data. Processing is conducted every 50 frames, and the length of one frame is 0.01 second.

5.2 Experimental Results

Fig. 1 shows that in both devices, the processing time of SSL and SSS with 60-channel data has been decreased significantly.

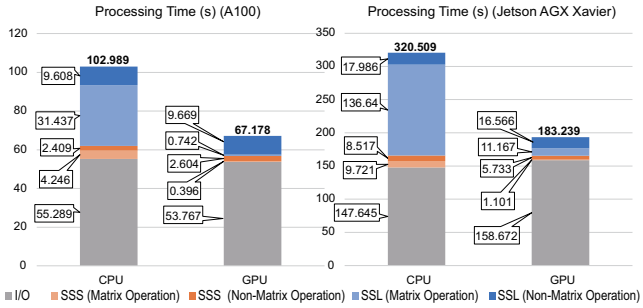

Fig. 1 Processing Time of SSL and SSS

A100 achieved 3.6× acceleration for SSL + SSS, while Jetson AGX Xavier is 5.0×. If we only focus on parts with matrix operations, A100 achieved 31.3× acceleration, while Jetson AGX Xavier is 11.9×. We can also observe that even on a GPU designed for edge computing, the processing time for matrix operations is significantly faster than on a high-performance CPU.

6. Conclusion

This paper proposed a GPU-based approach for PyHARK acceleration. Experimental evaluations reveal that our approach effectively reduces the processing time of PyHARK on two different scales of devices, which also shows its potential for application in different scenarios. In future work, we plan to explore more optimal parallelization techniques, such as dynamic block size, and establish a unified architecture for PyHARK processing that includes FPGAs.

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] Kazuhiro Nakadai, Hiroshi G. Okuno, and Takeshi Mizumoto. Development, deployment and applications of robot audition open source software hark. *Journal of Robotics and Mechatronics*, 29(1):16–25, 2017.

[3] Kazuhiro Nakadai, Katsutoshi Itoyama, and Masayuki Takigahira. Pyhark: Hark python package supporting online and offline processing. *SIG-Challenge*, 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.