Accelerate Hough Transform with Multi-GPUs

官澔恩 楊卓敏 邱信瑋 蓋彥文
R12922060 R12944068 R12922054 R12922183

This report does not have any double assignment or any completed work before this semester.

***Abstract*— In the context of advancing autonomous driving technology, efficient lane detection is critical. This research focuses on accelerating the Hough Transform, a fundamental method for lane detection, using multi-GPU platforms. Building on an existing open-source single-GPU implementation, we aim to enhance computational efficiency through frame size reduction, frame splitting, and accumulator splitting strategies. Our experiments revealed that frame splitting strategies could not significantly improve performance due to data transfer time. On the other hand, balanced accumulator splitting strategies with asynchronous function calls significantly contributed to performance improvement. Eventually, our best optimization achieved a 1.46x speedup compared to baseline work. This study underscores the importance of balancing computational and data transfer costs in multi-GPU accelerated algorithms for the Hough Transform. Our open-source work is available at https://github.com/Alan-Kuan/NTU-IIV-Final-Project.**

1. Introduction

Nowadays, with the booming electric vehicle market and increasingly mature autonomous driving technology, there are higher demands for speed and efficiency. Among these, lane detection technology plays a crucial role, possessing significant research and practical value. Therefore, we have decided to focus on researching and advancing lane detection technology. Hough Transform is a famous lane detection method which introduced in the course this semester, therefore, we decided to perform acceleration with GPUs on it.

Hough Transform is a technique which can be used to identify features of a particular shape within an image. Because it requires that the desired features he specified in some parametric form, the classical Hough Transform is most commonly used for the detection of regular curves such as lines, circles, ellipses, etc. The main advantage of the Hough Transform technique is that it is tolerant of gaps in feature boundary descriptions and is relatively unaffected by image noise. The simplest case of Hough Transform is detecting straight lines. In general, the straight line $y = mx + b$, can be represented as a point (b, m) in the parameter space. However, vertical lines pose a problem. They would give rise to unbounded values of the slope parameter m. Thus, for computational reasons. Duda and Hart [1] proposed the use of the Hesse normal form

$$\rho = x \cos\theta + y \sin\theta,$$

where ρ is the distance from the origin to the closest point on the straight line, and θ is the angle between the x axis and the line connecting the origin with that closest point.

After looking for related works, we found an open-source Hough Transform GPU acceleration implementation on GitHub [2] as the previous work we referenced. In this work, firstly, images are read from the video frame by frame, and the white and yellow lane markers are filtered from the image, followed by a series of pre-processing steps, such as Gaussian blur, Canny edge detection, masking region of interest, etc. Secondly, The Hough Transform is implemented using two primary CUDA kernels, one is responsible for trying θ in range of $45° \pm 16°$ and $135° \pm 16°$ and obtain their corresponding ρ for non-zero pixels and adding votes to an accumulator buffer, the other is for finding local maxima that reach a threshold based on the number of votes for every ρ and θ combination. The local maxima of votes represent the notable lines in the original frame. Finally, the lines are drawn onto the frame and displayed in the output video.

However, It was achieved with only single GPU . Thus, our goal is to enhance the computational efficiency and run it on a platform with two GPUs for further acceleration.

2. Methodology

As illustrated in Figure 1, it is the overall workflow of our program. The principal aim is to mitigate computational complexity and efficiently allocate tasks across multiple GPUs, thereby optimizing performance and achieving computational efficiency. In Section 2.1, we introduced how to determine the region of interest (RoI), which was the only pan copied to the GPU. Subsequently, in Section 2.2. we presented methods for evenly distributing the RoI frame across multiple GPUs for performing the Hough Transform. Once each GPU completed its task and obtained Its respective results, we performed an *All Reduce* operation via NCCL on all the results. Eventually, in Section 2.3 we put forward strategies for how the accumulator efficiently identified notable lines that we represented.

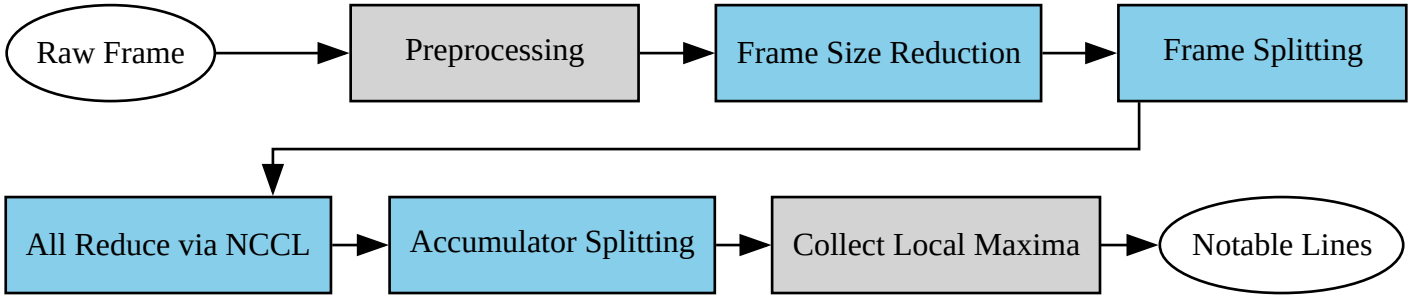

Figure 1: Overview of the Execution Flow

2.1. Frame Size Reduction

In each frame, only a particular region is crucial to lane detection, as the green region shown in Figure 2. This region of interest (RoI) can comprehensively encompass the lane in which the vehicle is currently traveling. Despite that the original method has already masked areas outside the RoI, the entire frame was still transferred to the GPU's global memory each time for Hough Transform computations. This process is evidently suboptimal, as only the RoI is necessary for the calculations. Consequently, we employ the *cudaMemcpy2D* function from the CUDA Runtime API to transfer only the frame's RoI into the GPU's global memory. This optimization is anticipated to enhance performance by reducing the volume of data copied to the GPU's global memory and eliminating superfluous computations.

2.2. Frame Splitting Strategies

The first GPU workload is to transform the input frame into an accumulator. To distribute the workload to 2 GPUs, we tried 4 different splitting strategies, including splitting the cropped frame into left-half and right-half, splitting the cropped frame into top-half and bottom-half, splitting the cropped frame column-by-column, and splitting the cropped frame row-by-row. Figure 3 visualizes these splitting strategies respectively.

Why we split the cropped frame into 2 bulk halves was because It is an intuitive way to distribute workload. This way, we could observe the basic performance of such simple methods. Besides, we expected the strategy which splits the cropped frame into left-half and right-half would outperform the one which splits the cropped frame into top-half and bottom-half. It is because the scene of lanes is usually bilaterally symmetric, resulting in a more balanced workload distribution.

Furthermore, to make the distribution even balanced, we split the cropped frame either column-by-column or row-by-row and then assigned even rows / even columns to GPU 0 and odd rows / odd columns to GPU 1. We expected to see the strategy which splits the cropped frame row-by-row could further beat the strategy which simply splits the cropped frame into left-half and right-half.

Figure 2: Illustration on Region of Interest

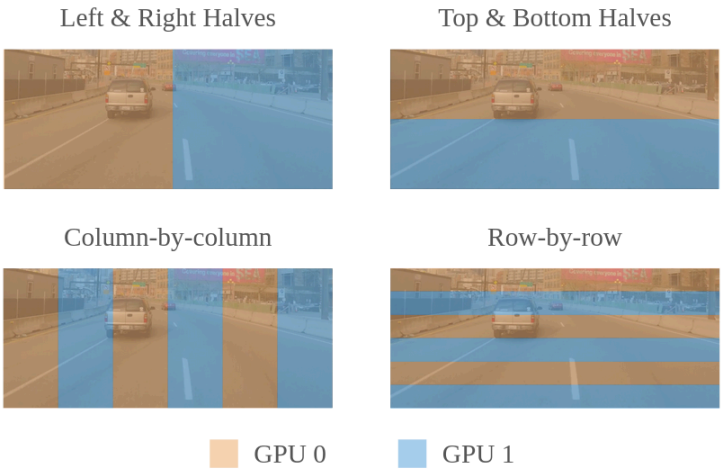

Figure 3: Illustration on Different Frame Splitting Strategies

2.3. Accumulator Splitting Strategies

After the accumulators from both GPUs were summed up into a single one via *All Reduce* provided by NCCL, we had to again distribute workload to both GPUs, instructing them to find local maxima on the complete accumulator, which represents notable lines in the original frame. The original work only considers cases with θ in the ranges of $45° \pm 16°$ and $135° \pm 16°$. As a result, in Figure 4, we can see there is a blank region in the middle of the accumulator.

Consequently, it is reasonable to split the workload vertically. This time, we only tried 2 different strategies to distribute workload, which are splitting the accumulator into left-half and right half, and splitting the accumulator pixel-by-pixel (since the width of accumulator is even, the effect is equal to column-by-column). These 2 strategies are illustrated in Figure 4.

We expected the latter would outperform the former because its workload distribution is more balanced. Worth mentioning, both GPUs would have the final accumulator in their own memory after the *All Reduce* operation, so we only had to tell both GPUs the indices they were responsible for.

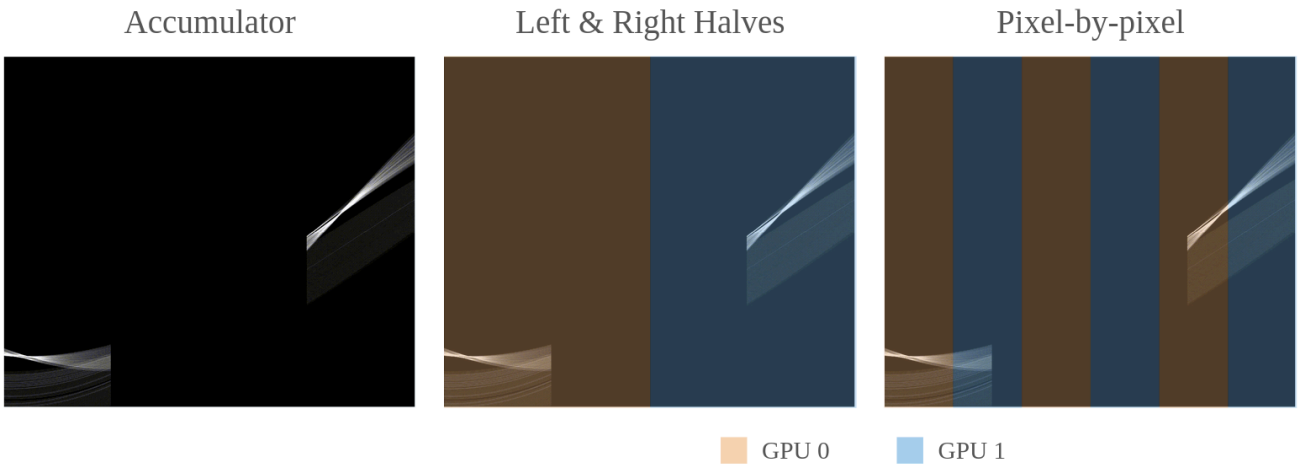

Figure 4: Example of an Accumulator and Illustration on Different Accumulator Splitting Strategies

3. Experiment Results

Our experiment environment was described in Section 3.1. The result of the final version of our work was presented first in Section 3.2. To provide a more detailed overview of our experiments, Section 3.3 presented the performance improvement by frame size reduction only using single GPU. In Section 3.4, we dived into the performance evaluation using synchronous calls with Left-Right-Distributed Accumulator. Section 3.5 presented a significant performance boost using asynchronous calls with Pixel-by-Pixel-

Distributed Accumulator. People might be curious about whether the performance boost was mainly due to asynchronous calls or Pixel-by-Pixel-Distributed accumulator. The answer is that the improvement is primarily due to asynchronous calls, and the analysis is conducted in Section 3.6.

3.1. Experiment Settings

The hardware and software configurations that were used for our work were listed in Table 1. To facilitate efficient communication between two GPUs, NVLink should be chosen as the communication protocol. However, the latest NVIDIA RTX4000 series GPUs does not support NVLink. Therefore, we decided to use two RTX2080ti GPUs in our experiment environment.

In addition, in the following experiments, we executed the program five times with each mentioned method, and averaged execution time of Hough Transform part of the program as a metric of performance.

Name	**Attribute**
CPU	AMD Ryzen Threadripper 3970X 32-Core Processor
GPU	NVIDIA GeForce RTX 2080ti * 2
NVLink Version	2.0
NVLink Bandwidth	25.781 GB/s
CUDA Version	12.3
NCCL Version	2.12.7

Table 1: The Hardware and Software Configuration

3.2. Different Strategies of Final Version

The performance of the final result of our work is presented in Table 2. We compared the execution time between 4 frame splitting strategies mentioned in Section 2.2: left and right halves (LR), top and bottom halves (TB), column-by-column (C-by-C) and row-by-row (R-by-R), and the original work, which implements Hough Transform with only one GPU. In Table 2, we can observe that our work achieves the overall speedup of 1.44x compared to the baseline. Notably, the LR splitting strategy demonstrated the best performance, with speedup of 1.46x.

	Baseline	**LR**[1]	**TB**[2]	**C-by-C**[3]	**R-by-R**[4]
Execution Time	1.321 s	0.904 s	0.929 s	0.920 s	0.919 s
Speedup	1.00x	**1.46x**	1.42x	1.44x	1.44x

[1] LR: left & right halves, [2] TB: top & bottom halves, [3] C-by-C: column-by-column, [4] R-by-R: row-by-row

Table 2: Performance Evaluation on Different Strategies of the Final Version

There were several improvements to achieve the final result, and each of them would be discussed later, and we would provide a detailed comparison in the following sections.

3.3. Frame Size Reduction Only

In this section, we discuss the performance comparison between the baseline and the version with frame size reduction only we have proposed. The experiment result is provided in Table 3. The version with frame size reduction only had a 1.10x speedup compared with the baseline, which was consistent to what we expected.

	Baseline	Size Reduction Only
Execution Time	1.321 s	1.196 s
Speedup	1.00x	**1.10x**

Table 3: Performance Evaluation of the Frame Size Reduction with Single GPU

3.4. Synchronous Calls with Left-Right-Distributed Accumulator

Starting from this section, the experiments are performed with two GPUs. The technique of frame size reduction mentioned in Section 2.1 was also adopted here, and performance evaluation of the baseline, frame size reduction with single GPU and the four frame splitting strategies with two GPUs mentioned in Section 2.2 is shown in Table 4.

	Baseline	Size Reduction Only	LR[1]	TB[2]	C-by-C[3]	R-by-R[4]
Execution Time	1.321 s	1.196 s	1.122 s	1.095 s	1.254 s	1.312 s
Speedup	1.00x	1.10x	1.18x	**1.21x**	1.05x	1.01x

[1] LR: left & right halves, [2] TB: top & bottom halves, [3] C-by-C: column-by-column, [4] R-by-R: row-by-row

Table 4: Performance Evaluation of Synchronous Calls with Left-Right-Distributed Accumulator Versions

From Table 4, we can see that using the LR and TB strategies with frame size reduction in two GPUs would perform better than only using frame size reduction in single GPU, since only half of the cropped frame was copied to each GPU. However, C-by-C and R-by-R strategies performed worse. Actually, in those 2 cases, the whole cropped frame was copied to each GPU, and we used *blockIdx* in CUDA kernel to tell each GPU which columns or rows it was responsible for. This was because it was costly to copy multiple interleaving columns or multiple interleaving rows to each GPU at a time. Unfortunately, by doing so, there came another issue that idle time emerged for each GPU during computation, since works of some GPU kernel threads in one GPU were done by the other GPU, and these threads quit early. As a result, the performance of these 2 cases did not meet our expectations.

In this version, memory copies were done synchronously. After profiling with NVIDIA NSight Systems [3], we noticed that execution timelines of one particular part were not overlapped, but it was one followed by the other. It indicated that the operations should be done asynchronously to increase the parallelism. We would show the improvement in Section 3.5. Also, the workload distribution of finding local maxima in accumulator was unbalanced. This improvement is shown in Section 3.5 and Section 3.6.

3.5. Asynchronous Calls with Pixel-by-Pixel-Distributed Accumulator

In this section, the improvement achieved through asynchronous memory copying were presented. We observed that memory copying could be done asynchronously, i.e., just after calling *cudaMemcpyAsync* function, the host started copying data to GPU 0 and returned immediately. Therefore, the host could start copying data to GPU 1 without waiting for the previous memory copy to finish.

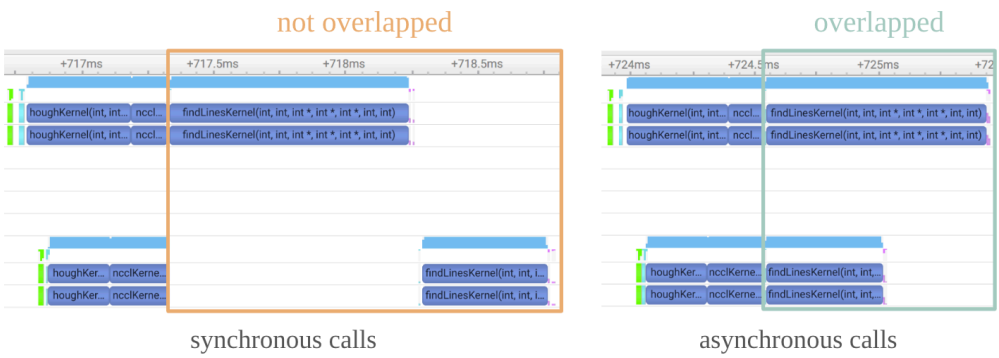

Figure 5: Impact of Synchronous Calls vs. Asynchronous Calls

From Figure 5, after replacing memory copying with asynchronous calls, the kernel function *findLinesKernel* could start simultaneously across two GPUs. This overlapping of data transfer processes allowed for more efficient utilization of computational resources and reduced overall memory transfer time, contributing to a notable performance enhancement.

	Baseline	**Size Reduction Only**	**LR**[1]	**TB**[2]	**C-by-C**[3]	**R-by-R**[4]
Execution Time	1.321 s	0.846 s	0.904 s	0.929 s	0.920 s	0.919 s
Speedup	1.00x	**1.56x**	1.46x	1.42x	1.44x	1.44x

[1] LR: left & right halves, [2] TB: top & bottom halves, [3] C-by-C: column-by-column, [4] R-by-R: row-by-row

Table 5: Performance Evaluation of Asynchronous Calls with Pixel-by-Pixel-Distributed Accumulator

Table 5 showed the performance evaluation of asynchronous call with balanced accumulator distribution over two GPU illustrated a significant boost compared to the baseline, as we previously anticipated. However, during our experiment, we found that the version with only frame size reduction over single GPU had a better performance in comparison with the versions with two GPUs. We speculated that the above phenomenon resulted from the communication overhead over two GPUs being much larger than what we expected.

3.6. Left-Right-Distributed Accumulator v.s. Pixel-by-Pixel-Distributed Accumulator

The performance comparison between the strategy that distributes accumulator to left half and right half and the one that distributes it pixel by pixel are shown in Table 6. The experiment was performed using C-by-C and R-by-R splitting strategies with asynchronous function calls under an environment with two GPUs.

	Baseline	**LR-Dist. Acc.**[1]		**P-by-P-Dist. Acc.**[2]	
		C-by-C [3]	R-by-R [4]	C-by-C [3]	R-by-R [4]
Execution Time	1.321 s	0.950 s	0.929 s	0.920 s	0.919 s
Speedup	1.00x	1.39x	1.42x	**1.44x**	**1.44x**

[1] LR-Dist. Acc.: Left-Right-Distribution Accumulator, [2] P-by-P-Dist. Acc.: Pixel-by-Pixel-Distributed Accumulator, [3] C-by-C: column-by-column, [4] R-by-R: row-by-row

Table 6: Performance Evaulation between balanced and unbalanced accumulator

As demonstrated in Table 6, the latter version outperformed the former in both C-by-C and R-by-R splitting strategies. In Figure 6, we used NVIDIA NSight Systems to profile the execution timeline of the first frame of the video. This analysis outcomes indicate that applying the balanced accumulator version could save 109 µs.

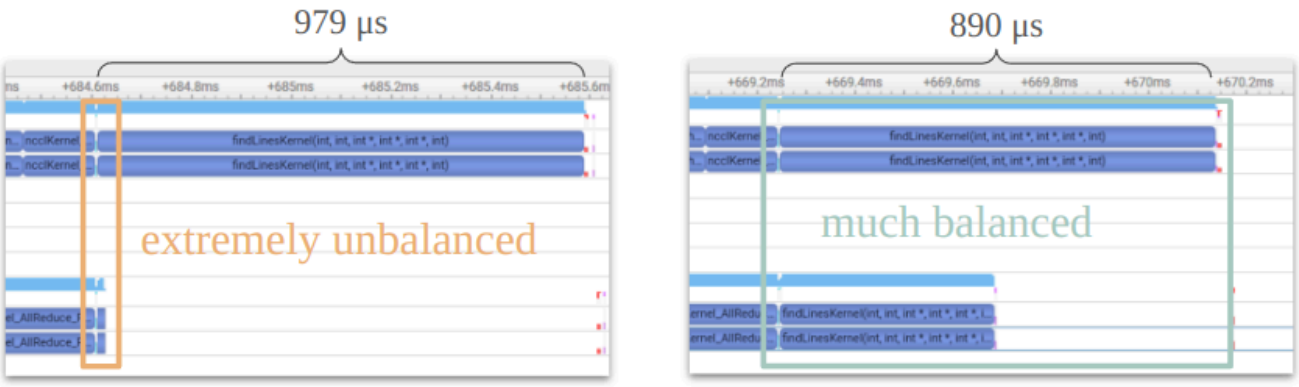

Figure 6: Impact of Unbalanced Workload vs. Balanced Workload

4. Conclusion

In this research, we aimed to accelerate the Hough transform by employing methods such as frame size reduction, frame splitting strategies, and accumulator splitting strategies. Our experiment results revealed that the frame splitting strategies did not show a significant improvement in performance. It was not due to the ineffectiveness of the strategies but the high cost of communication. The inherent computational complexity of the Hough Transform should theoretically benefit from workload distribution across multiple GPUs. Nevertheless, the pruning techniques by masking out RoI and enumerating only lines with θ between $45° \pm 16°$ and $135° \pm 16°$ substantially reduced the computational burden. Consequently, the time saved through splitting a frame to 2 GPUs was not significant. Moreover, the data transfer time required by NCCL to synchronize the results from different GPUs nearly offset the time saved by frame splitting strategies, leading to an insignificant overall efficiency gain.

The balanced accumulator splitting strategies with asynchronous calls demonstrated a notable improvement in overall performance. The time saved by utilizing 2 GPUs was significant, and unlike frame splitting strategies, accumulator splitting strategies did not require data transfer via NCCL for results synchronization. This contributed to a more substantial enhancement in overall computational efficiency.

In summary, compared to the baseline methods, our research achieved an overall speedup of 1.46x. This study highlights the importance of considering both computational and data transfer costs when designing multi-GPU-accelerated algorithms for the Hough Transform.

Acknowledgement

Our work originates from the work, “CUDA Lane Detection” (https://github.com/jonaspfab/cuda-lane-detection), by Jonas Pfab, which is open-sourced on GitHub.

Work Distribution

- 官澔恩：Project Initiator, Group Organization, Experiment Implementation, Slides Making and Report Writing
- 楊卓敏：Experiment Implementation, Slides Making and Report Writing
- 邱信瑋：Experiment Implementation, Slides Making and Report Writing
- 蓋彥文：Experiment Implementation, Slides Making and Report Writing

References

[1] R. O. Duda and P. E. Hart, “Use of the Hough transformation to detect lines and curves in pictures,” *Commun. ACM*, vol. 15, no. 1, pp. 11–15, Jan. 1972, doi: 10.1145/361237.361242.

[2] J. Pfab, “Jonaspfab/CUDA-lane-detection: Cuda implementation of a Hough Transform based lane detection algorithm.” [Online]. Available: https://github.com/jonaspfab/cuda-lane-detection

[3] N. D. (n.d.)., “Nvidia Nsight Systems.,” [Online]. Available: https://developer.nvidia.com/nsight-systems