Master's Thesis of Engineering

Parallelization of Autonomous Driving Tasks for Safe High-Speed Vehicle Control

안전한 고속 주행 제어를 위한 자율 주행 연산 작업의 병렬화

August 2019

Graduate School of Seoul National University

Department of Computer Science and Engineering

Alena Kazakova

Parallelization of Autonomous Driving Tasks for Safe High-Speed Vehicle Control

안전한 고속 주행 제어를 위한

자율 주행 연산 작업의 병렬화

지도교수 이 창 건

이 논문을 공학석사 학위논문으로 제출함

2019 년 06 월

서울대학교 대학원

컴퓨터공학부

알 료 나

알료나의 공학석사 학위논문을 인준함

2019 년 07 월

위 원 장 _____ 이 상 구 _____ (인)

부위원장 _____ 이 창 건 _____ (인)

위 원 _____ 이 재 진 _____ (인)

Abstract

Parallelization of Autonomous Driving Tasks for Safe High-Speed Vehicle Control

Alena Kazakova

Department of Computer Science and Engineering

The Graduate School

Seoul National University

Autonomous driving systems have strict performance requirements in terms of both real-time constraints and safe operation. Defining these requirements, along with identifying potential bottlenecks and acceleration techniques, determines appropriate computational platform specifications for cost-effective system design. This work presents 1) how real-time constraints in autonomous driving determine the maximum safe vehicle speed, 2) acceleration techniques to meet these constraints, 3) cost/performance trade-offs.

We discuss several combinations of possible optimizations and their trade-offs in terms of their effects on response time and computational requirements. We obtained experimental results from the lane following algorithm executed on a scaled autonomous driving platform. We reduced latency and increased throughput of the algorithm by utilizing parallelism and pipelining techniques, as well as GPU acceleration with CUDA, and achieved stable vehicle performance at high speed.

Keywords: autonomous driving, real-time constraints, parallel programming

Student Number: 2017-21749

Table of Contents

List of Figures

Chapter 1. Introduction

Building autonomous driving systems is very challenging due to strict performance requirements. These systems must always make safe operational decisions to avoid accidents, which is why they use computationally intensive advanced machine learning and computer vision algorithms to deliver high precision. Despite a large amount of computation, it is critical for the system to react to the traffic conditions at real-time, which means the end-to-end processing always needs to finish at a strict deadline. The processing latency must be less than 100 milliseconds for the system to be able to react to constantly changing traffic conditions faster than a human driver. The sensor frame rate must be higher than 10 frames per second [1] in case the traffic conditions change drastically between two neighboring frames.

Many engineers do not consider real-time constraints and try to increase performance by investing in more powerful processing units and hardware accelerators or by using software libraries which require such hardware. This investment allows applications to get faster without having to modify them or the libraries that they rely on. However, this approach is not cost-effective.

To investigate this question, we use our previously built scaled autonomous driving platform [2], that has primarily benefited from the related work done in the community [3] [4] [5], to showcase the effects of real-time constraints on the maximum safe vehicle speed. Based on that, we formalize the computational requirements and present how to achieve them by accelerating autonomous driving tasks.

In this thesis, we want to keep the computational pipeline simple and focus on a vision-based autonomous driving system that only uses a camera for perception. It is more widely used in the industry, as opposed to a LIDAR (Light Detection and Ranging) based system due to its high cost and the fact that it utilizes algorithms that are difficult to parallelize [6].

For our case study, we chose to implementation the lane following algorithm because it can demonstrate the minimal requirements to build an end-to-end autonomous driving system. In this architecture video captured by a camera is streamed into a processing engine, which performs lane detection, vehicle localization, path prediction, and vehicle control.

Computer vision is devoted to analyzing, modifying, and high-level understanding of images. Its objective is to "understand" the environment and use that to control a robotic system. However, computer vision is computationally expensive. Many computer vision scenarios are time-critical, which implies that the processing of a single frame should complete within 30-40 milliseconds. This requirement is very challenging, especially for embedded architectures. Often, it is possible to trade off quality for speed. To meet the constraints of time and the computational budget, developers either compromise on quality or invest more time into optimizing the code for specific hardware architectures. [7]

Profiling is required to understand system requirements and to find bottlenecks to avoid performance issues. We found bottlenecks of our algorithm and implemented an acceleration framework based on manager/worker and pipeline models for multithreading. We reduced latency and increased throughput of the algorithm, by utilizing parallelism and pipelining techniques, as well as GPU acceleration with CUDA. We discuss several combinations of possible optimizations and their trade-offs in terms of their effects on reaction time, computational resource requirements, and maximum safe vehicle speed.

The rest of the paper is organized as follows: section 2 describes our algorithm implementation and the real-time constraints, section 3 presents the acceleration framework and discusses the results, section 4 suggests improvements and future work, and finally, section 5 concludes the paper.

Chapter 2. Case Study: Lane Following

2.1. Lane Detection Algorithm

We implemented a lane detection algorithm following Advances Lane Finding project from Udacity's Self Driving Car Engineer Nanodegree Program [8]. We used OpenCV [9], which stands for Open-Source Computer Vision and contains extensive libraries of functions for computer vision work.

In this algorithm we first transform image for lane lines to appear parallel from a bird's eye view perspective, then we apply a combination of color and gradient thresholds to create a binary image where the lane lines are clearly visible, and finally, we locate the lane lines using the histogram and sliding window search approaches.

2.1.1. Perspective Transform

Perspective transform maps the points in a given image to different desired image points with a new perspective. We transform an image to a bird's eye view perspective by manually choosing four source points lying along the lines that define a trapezoidal shape on an image with straight lane lines. Also, we select four destination points that represent a rectangle on a warped image where lines should appear straight and vertical. We then compute the perspective transform and use it to warp an image.

2.1.2. Color and Gradient Threshold

Various color thresholds can be applied to find the lane lines in images. We found that Red channel of BGR (Blue, Green, Red) color space and Saturations channel of HLS (Hue, Saturation, Lightness) color space do a reasonable job of highlighting the lines. The Red channel does a better job at picking white lines, while Saturation channel performs best under very different color and contrast conditions.

Gradient threshold can also be used to find lane lines in images. Applying the Sobel operator to an image is a way of taking the derivative of the image in the x or y direction to detect edges.

The operators of $Sobel_x$ and $Sobel_y$:

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, S_x = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

These operators take the derivative by overlaying either one on a 3 x 3 region of an image. If there is little change in values across the given region, then the sum of the element-wise product of the operator and corresponding image pixels will be zero, implying a flat gradient.

$$gradient_x = \sum (region \times S_x)$$

Taking the gradient in the x-direction emphasized edges closer to vertical, and the gradient in the y-direction emphasized edges closer to horizontal.

Sobel accepts a single-color channel, which we found works best with Lightness channel of HLS color space. Since x-gradient does a cleaner job of picking up the lane lines, we apply the $Sobel_x$ operator to the image and then take the absolute value of it and convert it to 8-bit. We then create a binary threshold to select pixels based on gradient strength.

We combine the threshold of Lightness channel's x-gradient with thresholds of the Red channel and Saturation channel to make the most robust identification of the lines.

2.1.3. Detecting Lane Lines

After applying perspective transform and thresholding, we have an image where the lane lines stand out clearly. We need to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line.

We take a histogram which adds up the pixel values along all the columns in the lower half of the image. In our thresholded image, pixels are either 0 or 255, so the

two most prominent peaks in this histogram are good indicators of the x-position of the base of the lane lines. We use that as a starting point for determining where the lane lines are, and then use sliding windows, placed around the line centers, to find and follow the lines up to the top of the image, with the given window sliding left or right if it finds the mean position of activated pixels within the window to have shifted. After we found all pixels belonging to each line, we fit a second-degree polynomial to all the relevant pixels we found in a sliding window. For a lane line that is close to vertical, we can fit a line using this formula:

$$f(y) = Ay^2 + By + C,$$

where A, B, and C are coefficients

A is the curvature of the lane line, B is the heading or direction that the line is pointing, and C is the position of the line based on how far away it is from the very left of an image ($y = 0$).

Fitting polynomial coefficients is achieved by solving the linear least-squares equation:

$$a = (X^T X)^{-1} X^T y,$$

where X is a Vandermonde matrix

We keep track of a previously detected lane by recording the lane width and the following characteristics of each line: if it was line detected, x-position of the base, polynomial coefficients, and a tangential angle of a curve.

We use tangential angles of curvature to see if right and left lines are parallel, and it can be calculated as follows:

$$\theta = \tan^{-1} \frac{x_0 - x_{img.rows/3}}{y_0 - y_{img.cols/3}}$$

If lines are not parallel, one of them was not detected correctly, so we choose the best one among them by comparing them with previously found lines. Then, we can approximate the missing line by using the lane width.

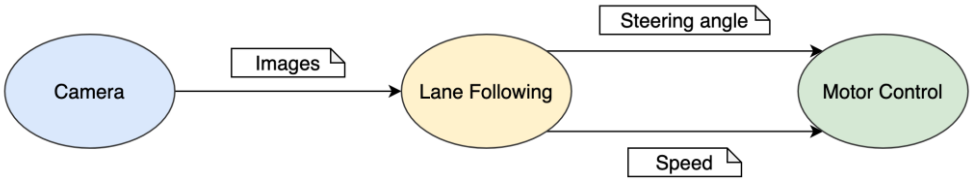

Figure 1. Computation graph of the end-to-end autonomous driving pipeline

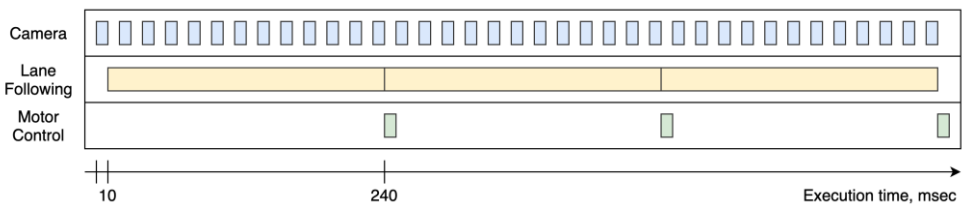

Figure 2. Gantt chart demonstrating the bottleneck of the initial implementation

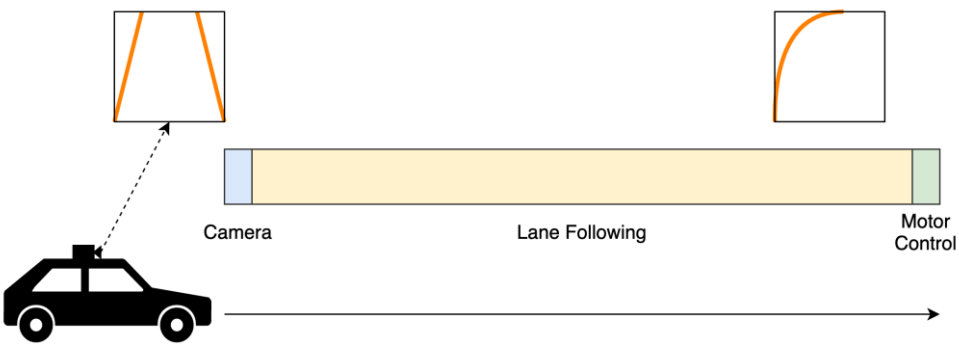

Figure 3. Unstable vehicle control scenario at high speed

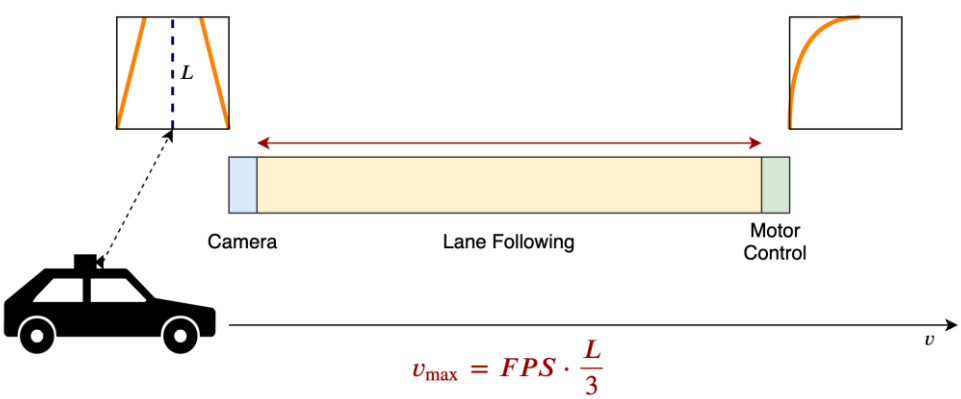

Figure 4. Algorithm acceleration and maximum speed bound for stable vehicle control at high speed

2.2. Determining Vehicle Position

We calculate the offset of the lane center from the vehicle position based on pixel values. Vehicle center coincides with the bottom center of the image as the camera is mounted at the center of the car and the lane center is the midpoint at the bottom of the image between the two detected lane lines. To convert from pixels to real-world measurements, we measured physical lane width and length in the camera's field of view and relevant pixels in x and y dimensions in perspective transformed image.

2.3. Real-time Constraints

While testing our initial implementation of the lane following algorithm on the scaled autonomous driving platform, we noticed that the performance becomes unstable at higher vehicle speed. The lane following algorithm takes image stream from a camera as an input, performs image processing to detect lane lines in the current frame, calculates the distance of a car from the center of the lane, predicts the desired trajectory, and outputs actuation commands for the motor controller (**Figure 1**). Through profiling, we determined that the response time of the algorithm was not fast enough (**Figure 2**). Single frame processing took around 240 milliseconds. When the car was entering a turn, by the time the frame processing was done, the actuation commands became irrelevant, causing the car to miss the turn (**Figure 3**).

Following our observations, we define real-time constraints and then apply them to path planning and vehicle control:

$$T_{actuation} = N \times T_{frame\ processing},$$

where $N > 2$ – to avoid positive feedback.

To achieve stable vehicle control at high speed, we had to speed up the execution time of the algorithm and define the maximum speed bound (**Figure 4**):

$$v_{max} = FPS \times L/3$$

where $FPS = 1/T_{frame\ processing}$ and

L is the length of desired trajectory in a car's FOV

If we take an arbitrary destination point that lies on the desired trajectory, by the time a car starts processing the next frame, it might have already surpassed it. To avoid it, the minimal distance towards our destination must be within the time it takes to process 2 frames. However, our frame processing time is not constant, and other factors can also affect reaction time, such as the time it takes to physically perform motor control commands. Therefore, to find the maximum speed bound, we multiply the frame rate and the length of the desired trajectory divided by $N = 3$.

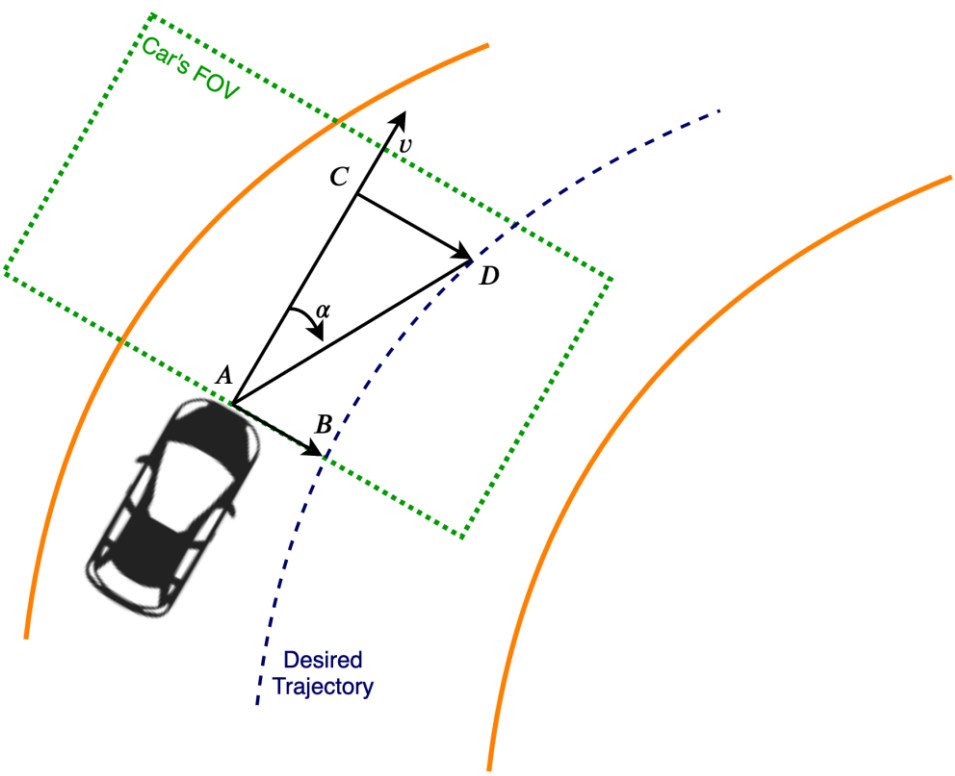

Figure 5. Path prediction and vehicle control model

2.4. Path Prediction and Vehicle Control

Our objective is to drive within a lane, keeping to its center when possible. Therefore, the desired trajectory is the centerline of the detected lane (**Figure 5**). To do so, we have to find a destination point on this line that satisfies the timing constraint. Therefore, we iterate through each point of the line until we reach the constraint:

$$T_{actuation} \times v_{pixels/sec} < \sum_{i=1} \sqrt{(x_B[i] - x_B[i-1])^2 + (y_B[i] - y_B[i-1])^2}$$

With coordinated of a destination point D, a vehicle position A and its offset from the lane center B, we can compute the steering angle α:

$$|CD| = x_D - x_A,$$

$$|AC| = y_D - y_B,$$

$$\alpha = \tan^{-1}\left(\frac{|CD|}{|AC|}\right)$$

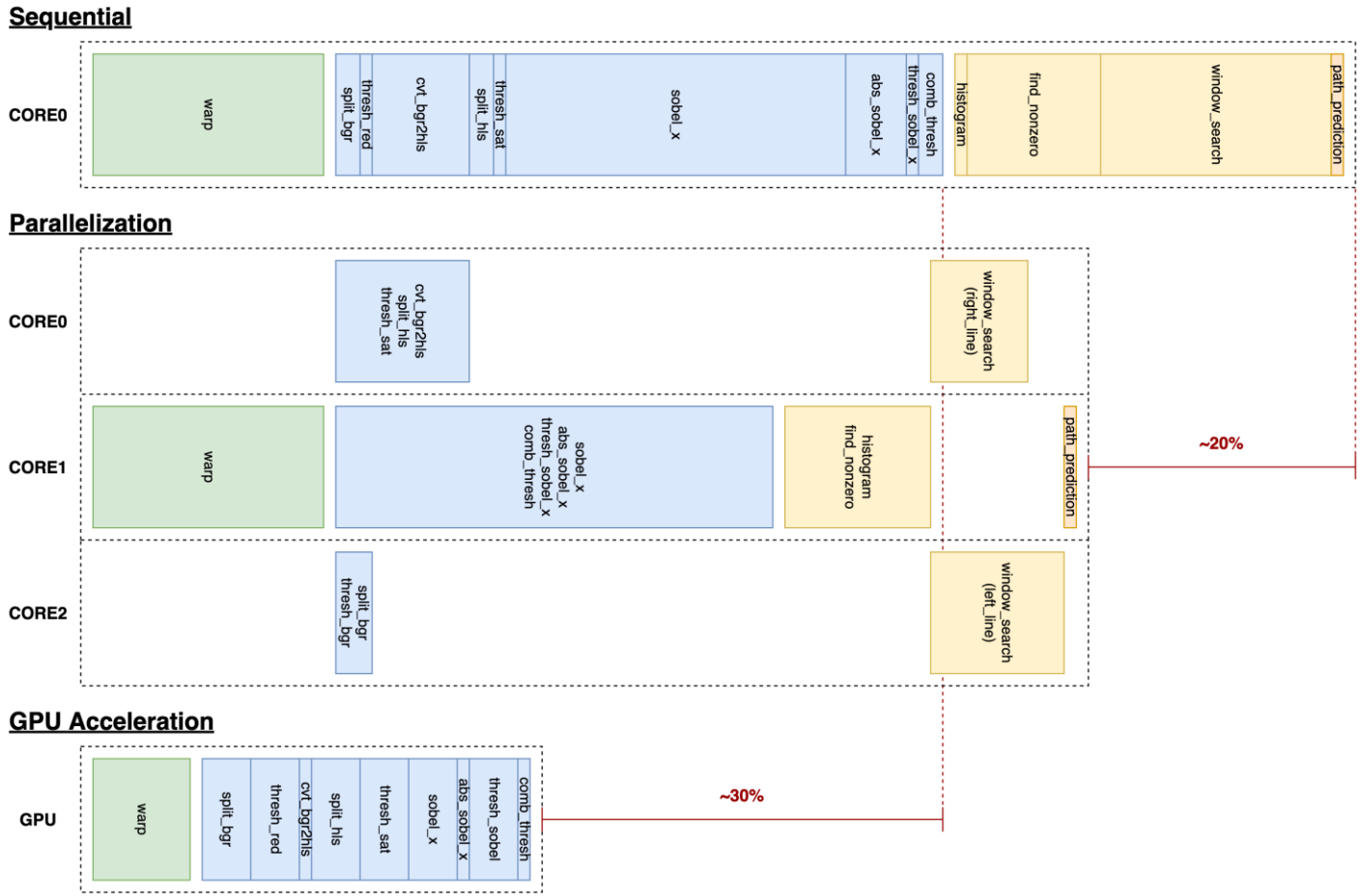

Figure 6. Identifying bottlenecks of the algorithm and conceptualizing optimization techniques

Chapter 3. Accelerating Autonomous Driving Tasks

3.1. Bottlenecks and Optimization Techniques

Task optimization requires an understanding of bottlenecks and data dependencies of the target. First, through time profiling, we identify the bottlenecks of the program. By knowing the flow of the program, we can separate it into independent blocks that can be executed concurrently, applying both function-level parallelization and data-level parallelization (**Figure 6**). For instance, thresholding channels of different color spaces are independent of each other and can be performed concurrently (function-level), as well as right and left lane line detection (data-level). Unfortunately, since the most computationally intensive blocks cannot be split any further, such parallelization technique only gives about 20% of the increase in performance. These blocks perform computer vision tasks, which are essentially operations on matrixes, and can be accelerated with GPU, which can speed up the execution time by about 30%.

Furthermore, pipelining can almost double the throughput of the program (**Figure 7**). However, these results could be improved if more blocks are accelerated with GPU and by balancing the pipeline.

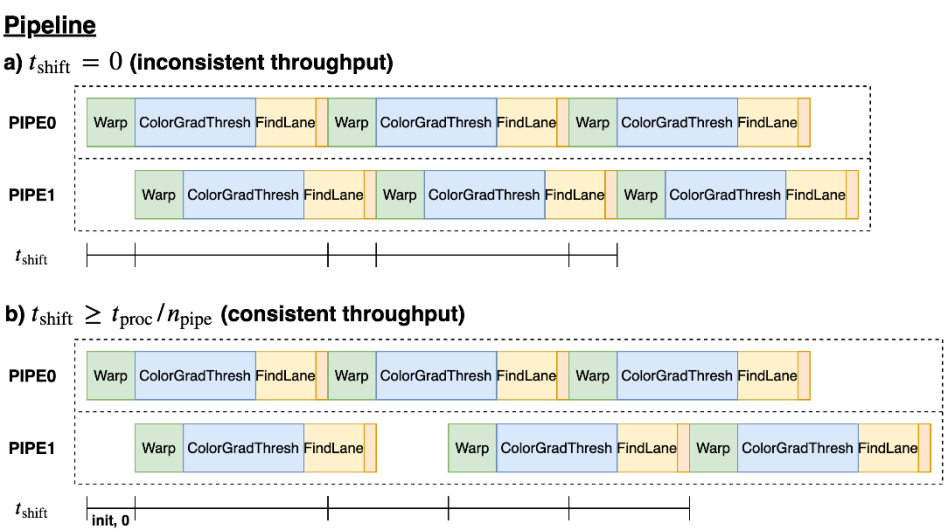

Figure 7. Pipeline-based concurrency

3.2. Acceleration Framework

We implemented an acceleration framework based on manager/worker and pipeline models for multithreading [10] (**Figure 8**). Our computational pipeline consists of three stages: Warp, ColorGradThresh, and FindLane that are executed in series, but concurrently, by different threads. Each stage is split into subtasks that have three states: initialized, running, and completed. Thread manager constantly checks if there are any initialized tasks, and if there is a free thread in the thread pool to perform it, then the task is transitioned into the running state. It also checks if there are any completed tasks to initialize the next task in line. When all tasks of the stage are completed, a thread worker sends a message to the thread manager. When the thread manager receives the message of stage completion, it passes its results to the next stage and sends a signal to a thread worker to initialize the first tasks of that stage.

Thread manager periodically starts the Warp stage and passes it the image frame from the camera. For the throughput consistency, we define the period:

$$t_{shift} \geq t_{frame\ processing}/n_{pipeline\ stages}$$

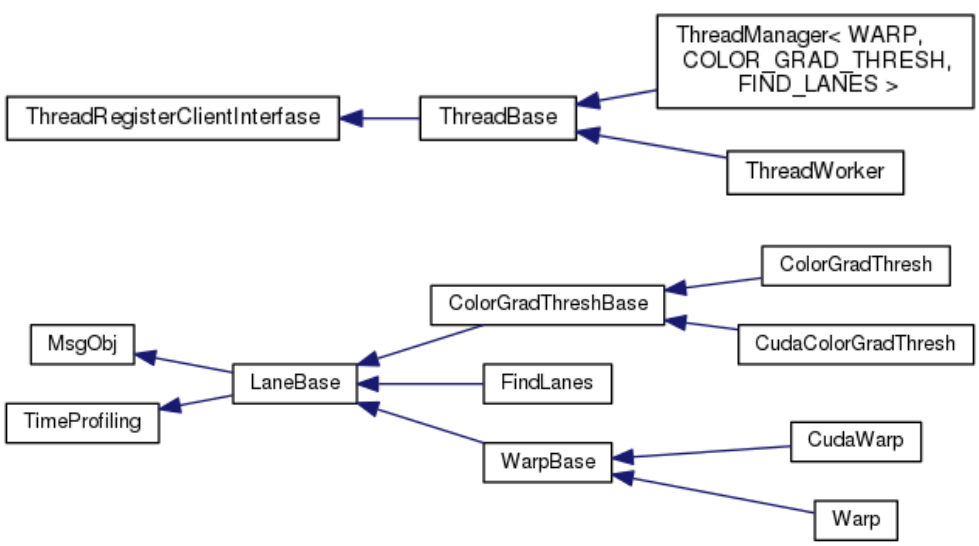

Figure 8. Implementation class hierarchy

After each FindLane is completed, the thread manager records its results and sends actuation commands to the motor controller. The last task in FindLane stage requires the results from the most recent completed FindLane. If there is a FindLane that reached the last task, but the preceding FindLane is uncompleted, the thread manager keeps it in the initialized state until the results are obtained.

3.3. Experiment Results

The experiments were conducted on a laptop with Intel Core i7-7700HQ processor, 16GB RAM and NVIDIA GeForce GTX 1050. Parallelization results gave about 10% speedup, and GPU acceleration gave about 20% speedup while pipelining almost doubled throughput (**Figure 9**). The obtained results are worse than our prediction due to thread manager and synchronization overheads, as well as the overhead of moving data between the GPU and CPU memories. Furthermore, GPU blocks the calling CPU task until it reaches synchronization point making it impossible for several threads to run GPU tasks at the same time. This affected the results of GPU acceleration when it was combined with parallelization or pipelining. This issue can be addressed by using different GPU streams and defining appropriate synchronization points [11] [12].

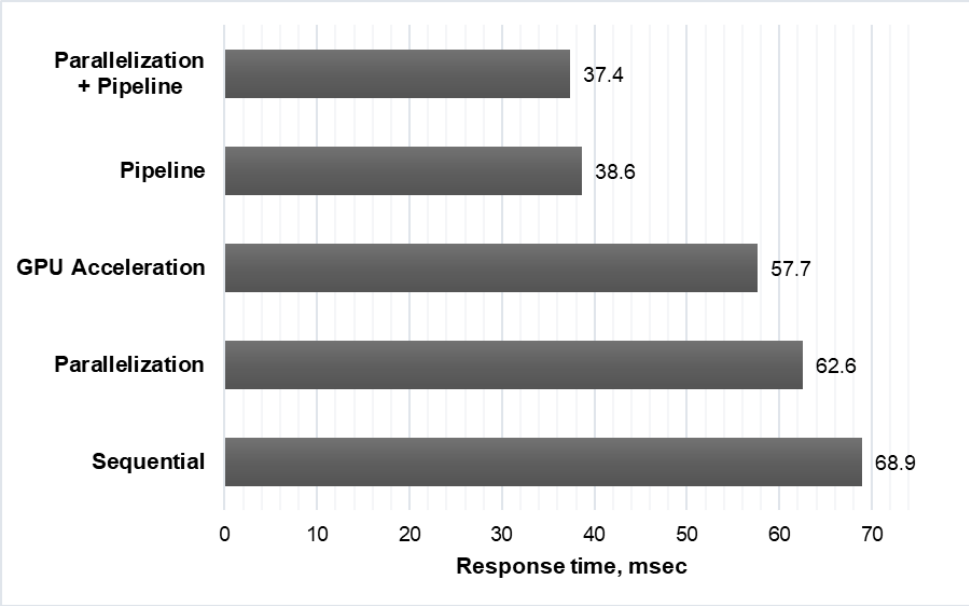

Figure 9. Summary of acceleration results

Our scaled autonomous driving vehicle uses NVIDIA Jetson TX2, which has ARM Cortex-A57 + NVIDIA Denver2 CPU and 256-core Pascal GPU [13]. The camera frame rate was set to 10 FPS. The initial implementation could stably perform only with 0.25 m/sec. The motor controller limit of our scaled autonomous vehicle is 0.75 m/s, which we could successfully achieve even after simply transitioning our code from Python to C++.

We found the maximum safe vehicle speed that can potentially be achieved by using each optimization technique (**Figure 10**). We also explored how the response time of the algorithm changed with the number of threads and pipeline instances (**Figure 11**). As expected, the more, the merrier. Considering the computational platform's specifications, we run tests up to 8 threads and 8 pipeline instances. However, we found that results with 4 threads and 4 pipeline instances were good enough. Therefore, it should be possible to use a less powerful computational platform. As for the GPU acceleration, it should not be necessary.

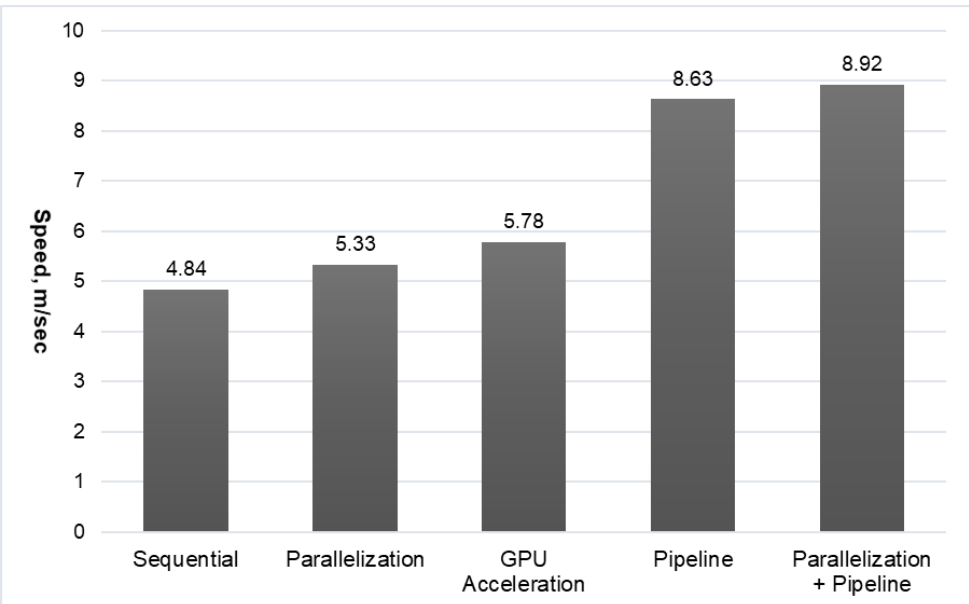

Figure 10. Maximum speed that can be achieved

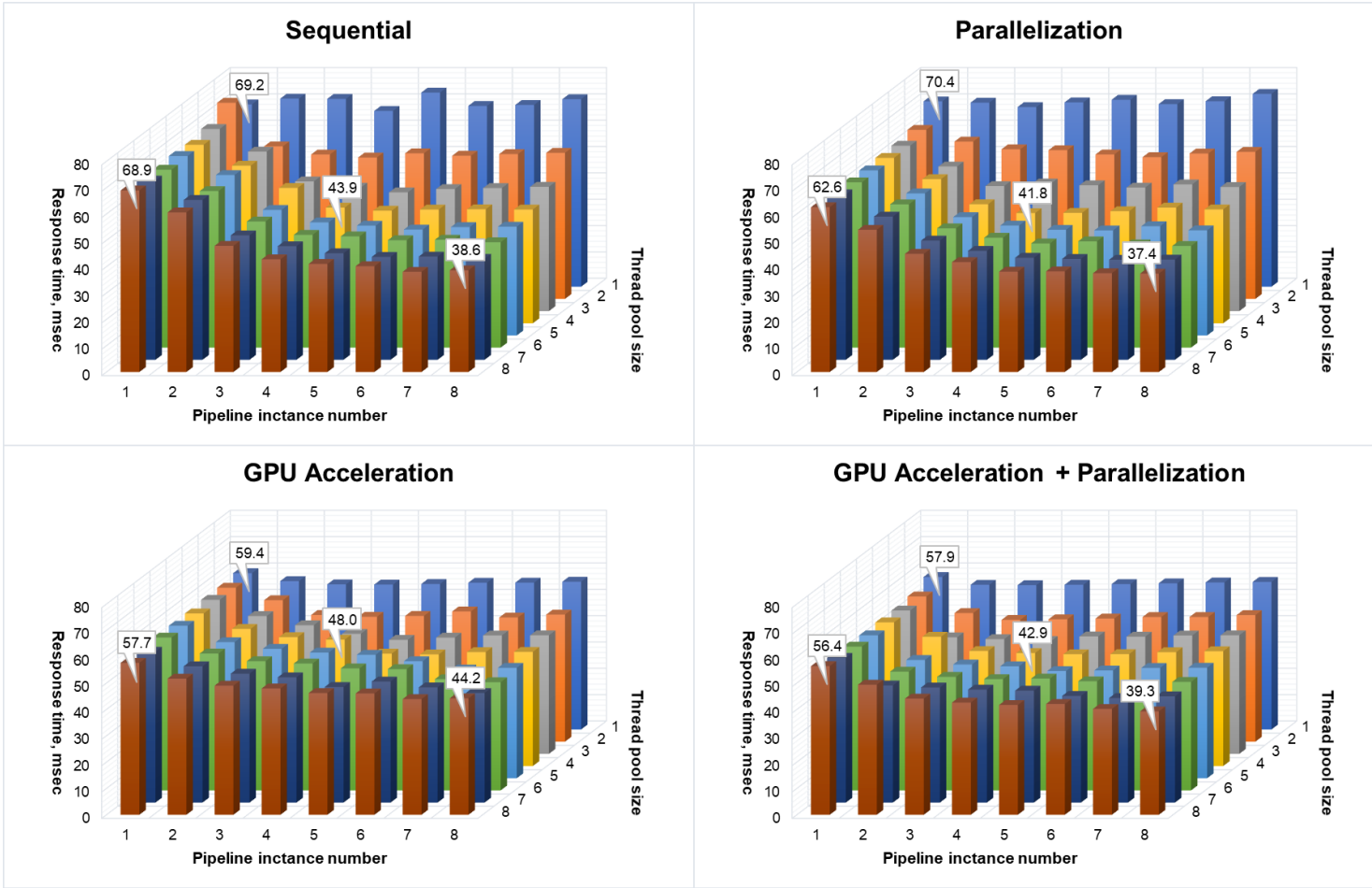

Figure 11. Effects of the number of threads and pipeline instances on response time

Chapter 4. Future Work

Generally, our case study algorithm is too naïve and not sufficient for autonomous driving in challenging conditions. It could benefit from more accurate calibration of color and gradient thresholding values, as well as coefficients for the pixel to meter and RPM to meters per second conversions. Source points for a perspective transform ideally should not be manually selected and could be chosen programmatically based on edge or corner detection and analyzing attributes like color and surrounding pixels. Window search can be improved by searching within a margin from previously found lines and averaging the results or by using more advanced techniques instead, such as applying convolution, which will maximize the number of "hot" pixels in each window. Kalman filter could improve path prediction, and PID controller could improve vehicle control.

The discussion of real-time constraints of autonomous driving tasks could be expended by looking at other algorithms.

Chapter 5. Conclusion

When developing autonomous driving systems, it is crucial to consider real-time constraints. If the reaction time of the system does not satisfy the requirements, performance becomes unstable. There are several ways of speeding up reaction time to satisfy real-time constraints: parallelization, pipelining, and GPU acceleration. Depending on the application, computational resource requirements must be taken into consideration.

We explored how the response time of lane following algorithm limited the maximum safe vehicle speed. Following our observations, we defined real-time constraints and improved the path prediction and vehicle control algorithm accordingly.

We found bottlenecks of our algorithm and accelerated it to meet the real-time constraints. We compared the effects of optimization techniques on reaction time, computational resource requirements, and maximum safe vehicle speed.

Bibliography

[1] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang and J. Mars, "The Architectural Implications of Autonomous Driving: Constraints and Acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Williamsburg, VA, USA, 2018.

[2] A. Kazakova, Y. Cho and C.-G. Lee, "OSCAR: An Open-Source, Self-Driving CAR Testbed," in *Korea Computer Congress 2018*, Seogwipo, Jeju-do, South Korea, 2018.

[3] M. O'Kelly, V. Sukhil, H. Abbas, J. Harkins, C. Kao, Y. V. Pant, R. Mangharam, D. Agarwal, M. Behl and P. Burgio, "F1/10: An Open-Source Autonomous Cyber-Physical Platform," *ArXiv*, vol. abs/1901.08567, 2019.

[4] S. Karaman, A. Anders, M. Boulet, J. Connor, K. Gregson, W. Guerra, O. Guldner, M. Mohamoud, B. Plancher, R. Shin and J. Vivilecchia, "Project-based, Collaborative, Algorithmic Robotics for High School Students: Programming Self-driving Race Cars at MIT," in *2017 IEEE Integrated STEM Education Conference (ISEC)*, Princeton, NJ, USA, 2017.

[5] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii and T. Azumi, "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, Porto, Portugal, 2018.

[6] S. Liu, J. Tang, Z. Zhang and J.-L. Gaudiot, "CAAD: Computer Architecture for Autonomous Driving," *ArXiv*, vol. abs/1702.01894, 2017.

[7] K. Pulli, A. Baksheev, K. Kornyakov and V. Eruhimov, "Realtime Computer Vision with OpenCV," *Queue*, vol. 10, no. 4, pp. 40-56, 2012.

[8] Udacity, "Self Driving Car Engineer Nanodegree," 2019. [Online]. Available: https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013.

[9] OpenCV, "OpenCV," 2019. [Online]. Available: https://opencv.org/.

[10] P. Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann Publishers Inc., 2011.

[11] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson and F. D. Smith, "Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, Dagstuhl, Germany, 2018.

[12] T. Amert, N. Otterness, M. Yang, J. H. Anderson and F. D. Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Paris, France, 2017.

[13] NVIDIA Corporation, "Jetson TX2," 2019. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/.

요약(국문초록)

자율 주행 시스템은 실시간 제약조건의 만족 및 안전한 차량 제어와 같은 엄격한 요구사항을 지닌다. 이러한 요구사항과 더불어 잠재적 병목 현상의 발견 및 적절한 가속화 기법 이 모두를 고려하여 적절한 자율주행 차량 플랫폼을 선택하는 것이 비용/효율 측면에서 바람직하다. 본 논문은 다음의 주제를 다룬다. 1) 실시간 제약 조건이 안전한 차량 제어를 보장하는 최대 속도에 미치는 영향 2) 실시간 제약 조건의 만족 아래 안전한 고속 주행 제어를 보장하는 차량속도 가속화 기법 3) 실험 결과 분석을 통한 비용과 성능 측면의 상충관계 제시.

본 논문은 멀티코어를 활용한 병렬화, 파이프라이닝 및 GPU 최적화 총 3 가지 기법들을 조합한 몇 가지 최적화 기법들을 제시하고, 개별 기법들의 실험결과를 반응 시간과 연산요구량 측면에서 분석하여 반응시간과 연산요구량의 상충관계를 구체적으로 제시한다. 본 논문은 간소화된 자율주행 플랫폼 상의 차선 추종 알고리즘을 기준 알고리즘으로 사용하였으며, 최적화 기법들을 적용한 실험 결과들을 통해 본 논문에서 제안한 기법이 지연시간 단축과 처리량 향상을 가져옴을 확인할 수 있다. 결론적으로, 본 논문에서 제시한 최적화 기법을 사용함으로서 고속에서도 안전한 차량 주행 제어가 가능함을 확인하였다.

주 요 어: 자율주행, 실시간 제한조건, 병렬 프로그래밍

학　번: 2017-21749