UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Real-Time Adaptation of Visual Perception

Permalink
https://escholarship.org/uc/item/8jq603hz

Author
Rathi, Avadhesh

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Real-Time Adaptation of Visual Perception

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Electrical Engineering

by

Avadhesh Rathi

December 2022

Thesis Committee:

Dr. Hyoseung Kim, Chairperson
Dr. Daniel Wong
Dr. Chengyu Song

Copyright by
Avadhesh Rathi
2022

The Thesis of Avadhesh Rathi is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my advisor, without whose help, I would not have been here. His constant support and guidance helped me accomplish this work.

To my parents for all the support.

ABSTRACT OF THE THESIS

Real-Time Adaptation of Visual Perception

by

Avadhesh Rathi

Master of Science, Graduate Program in Electrical Engineering
University of California, Riverside, December 2022
Dr. Hyoseung Kim, Chairperson

Autonomous driving has taken a leap in recent years due to the significant improvements in convolutional neural networks and advanced video processing algorithms. Despite these advancements, the criticality of the application has been of major concern as any error can lead to loss of life. When designing an autonomous vehicle, amongst all the different stages of perception, planning and control, perception takes the most amount of time. Understanding the scene accurately and in time is important and has been a challenge due to computationally heavy algorithms and machine learning models used. Recent studies have focused on this issue and proposed various approaches that utilize multiple sensors and expensive setups for perception. However, these systems do not adapt and scale to utilize the underlying resources to their fullest.

In this thesis, we present a simple yet effective approach that focuses on reducing the latency for real-time visual perception in autonomous vehicles. We take input from a single camera and perform lane and object detection. By dividing the input frame into critical and non-critical regions and utilizing both CPU and GPU resources for the work-

loads, we obtain both fast and accurate results. In addition, we propose an adaptive scaling algorithm that tunes the input image resolution based on the processing time to ensure a real-time processing timeline. To test our approach, we build a small prototype of the car using Jetson Nano equipped with a wide-angle camera and a motor driver to control four DC motors on each wheel. We conduct a case study on this prototype using a real-world dataset and compare it against conventional approaches. The results suggest that our proposed system recognizes the objects in the frame with minimal latency and good accuracy as compared to the other approaches.

Contents

List of Figures x

List of Tables xi

1 Introduction 1
1.1 Motivation . . . 1
1.2 Thesis Outline . . . 3
1.3 Contribution . . . 3

2 Related Work 5
2.1 Small-scale cars . . . 5
2.2 Perception . . . 6
2.3 Latency and Accuracy . . . 8

3 System Design 10
3.1 Overall Design . . . 10
3.2 Perception Algorithms . . . 13
3.2.1 Lane Detection . . . 14
3.2.2 Object Detection . . . 16
3.3 Critical Region Identification . . . 18
3.4 Runtime Mechanism with Adaptive Scaling . . . 19
3.5 CPU and GPU workloads . . . 22

4 Evaluation and Results 24
4.1 Prototype Implementation and Testing . . . 24
4.2 Inference Timings . . . 26
4.3 Case Study . . . 28
4.3.1 Dataset Selection . . . 28
4.3.2 Evaluation Criteria . . . 29
4.3.3 Evaluation and Results . . . 30

5 Conclusions 35
5.1 Future Work . . . 36

Bibliography **37**

List of Figures

3.1 Hardware Block Diagram 11
3.2 Overview of System Flow 12
3.3 CNN Architecture of DAVE-2 Model [4] 15
3.4 Overview of YOLOv4 Object Detector [3] 17
3.5 Critial Region Identified for Steering Angle of 20°. [6] 19

4.1 Prototype of the Car 25
4.2 Testing the Car on Track 26
4.3 Frame of Sight (frame 250). [6] 31
4.4 Input Frame for First Detection in Approach 1. [6] 32
4.5 Input Frame for First Detection in Approach 2. [6] 32
4.6 Input Frame for First Detection in Approach 3. [6] 33
4.7 Input Frame for First Detection in Approach 4. [6] 34

List of Tables

4.1 Time and processing rates for YOLOv4 27
4.2 Results for the Case Study 34

Chapter 1

Introduction

1.1 Motivation

With recent advancements in the field of deep learning, there has been a surge in the number of smart and intelligent products around us. Throughout this period, the autonomous driving application has been and is still a very hot topic due to a variety of benefits: reduce traffic congestion, decrease accidents and save lives, increase safety and make transportation more accessible. The idea of driver-less cars has been a fascinating research topic [15, 2, 13]. And the increasing amount of computing power through various processing platforms like GPU (Graphics Processing Unit) and TPU (Tensor Processing Unit) has further boosted this. These units are capable of performing tons of operations in parallel and have dedicated architecture for machine-learning applications [22, 24, 23]. With all these advancements, perception has still been an area of concern in autonomous driving [10, 15]. Understanding and evaluating the scene around the car fast and accurately is critical to the application.

A ton of factors have to be taken care of when deciding the motion of the vehicle as it is very crucial to the safe and reliable operation of an autonomous vehicle [27]. Multiple sensors are used around the car to get a complete understanding of the scene. These sensors continuously capture information in various formats and are the primary decision-making factors. Most of the implementation relies on multiple sensors facing what is in front of the car to gather a 3D sense of nearby objects and understand them. These include expensive sensors like LiDAR (Light Detection and Ranging). In [14], the authors introduce various hardware support and computation algorithms for ADAS (Autonomous Driving Assistance System) and discuss the pros and cons.

Modern techniques are developing different image processing and computer vision mechanisms to compensate for the expensive LiDAR sensors without compromising accuracy but it has been challenging due to excessive latency. Perception has always been compute heavy but has various aspects to it. In this thesis, we present a fast and simple yet innovative approach to understanding the scene on embedded edge computing devices like Nvidia Jetson Nano. We focus on two important aspects of visual perception – lane detection and object detection – and propose an approach that concentrates on reducing the latency in perception without compromising the accuracy. By efficiently utilizing both CPU and GPU cores for processing different regions of the image based on their criticality, we attain this desired goal. We also introduce an adaptive scaling mechanism that automatically tunes the resolution of the input frames and finds the best-suited value for that particular computing platform while ensuring that processing rates are not compromised.

1.2 Thesis Outline

The thesis is organized into different chapters as follows:

- Chapter 2 discusses the previous work done in the domain of perception for autonomous driving applications. This chapter provides insights into different approaches already proposed by different authors and focuses on the benefits and drawbacks.
- Chapter 3 describes the entire system design of the thesis and a deep dive into each of the design choices made with the reasons behind them. It also presents the implementation from both hardware and software perspectives.
- Chapter 4 focuses on the evaluation criteria and methodology. It presents a case study using the KITTI dataset [6] to evaluate our proposed system and compare them against conventional approaches. Finally, the chapter discusses the results and draws a conclusion based on them.

1.3 Contribution

The main contributions of this thesis work are as follows:

- Propose a novel runtime mechanism that utilizes both the GPU and CPU resources of an embedded edge device efficiently to reduce the end-to-end delay of perception algorithms, without compromising on the accuracy.
- Propose a simple yet effective approach to identify the critical region in the scene.

- Propose an adaptive scaling algorithm that makes the system dynamically adjust its performance based on onboard computational capabilities.

Chapter 2

Related Work

In this chapter, we discuss previous research and studies in related areas and compare them against our approach. We divide the previous work into different sections based on the area of research.

2.1 Small-scale cars

There have been numerous research papers that center their work on building autonomous vehicles using cheap computing platforms and lightweight algorithms and machine learning models. The authors of [20] propose building a Level-4 autonomous driving vehicle using a single off-the-shelf card. Though they utilize low-cost boards like Jetson AGX Xavier, they get the input data from a cluster of expensive sensors like LiDARs. In [1], the authors propose a low-cost neural network-based car, called DeepPicar, which utilizes a Raspberry Pi and camera input for lane detection only. The paper conducts a thorough study on inferencing the lane detection model with different workloads but fails

to fully utilize the computing resources on the Raspberry Pi to include other important autonomous driving functionalities. In our proposal, we focus on low-cost embedded edge devices and present an approach for a fast and accurate perception in autonomous driving by utilizing all available onboard resources.

2.2 Perception

Various methods of perception have been developed over the years. Based on the level and scale of implementation, different techniques are used, ranging from convolutional neural networks to traditional image processing and computer vision mechanisms. [14] summarizes the different sensors and prediction algorithms used for perception. LiDARs have been very popular and advantageous for depth estimation. Authors in [9] propose a method to 3D object detection, classification and tracking using point clouds formed by LiDARs. Here, accuracy is of more interest than latency. In another research, DeepDriving [5], the authors propose mapping the input image to a small number of key perception indicators that directly relate to the affordance of road/traffic state for driving. Though the perception map generated shows good resemblance to the actual scene, it does not detect the type of object. Also, since the proposed CNN model uses the AlexNet architecture, it has around 62 million parameters that hinders a real-time processing.

In our implementation, since we use an edge device for processing, we focus on computationally lightweight methods. Also, we use only lane and object detection for the perception. One of the first neural network based lane detection mechanism was proposed by Pomerleau et al. [18]. Recent advancements have come up with more and more accurate

lane detections. For example, authors in [21] use an edge-cloud computing mechanism for fast and accurate results. For our implementation, we take the motivation from [1] and use the NVIDIA's DAVE-2 model whose CNN architecture is originally proposed in [4]. There are 9 layers, including 5 convolutional layers and 3 fully connected layers. Calculating the total number of parameters/weights, it comes to be around 250,000 which computes fast on edge devices. In addition, this model has been tested on NVIDIA's real autonomous driving car as mentioned in [4].

Object detection is an important part of perception and detections from single image has evolved with the advancements in machine learning. The various object detection algorithms are proposed to perform multiple detection from the single image, the algorithm - region based CNN (R-CNN) [8] divides the whole input image into 2000 sub-regions based on selective search algorithm and pass each region into the CNN model to extract feature which is further given to output dense layer to predict its class and it also predicts four values which correspond to the bounding box of the object in an image. Next, fast R-CNN [7] was proposed to tackle the limitation of R-CNN which takes a long time to classify 2000 regions from a single image. To reduce the time taken we pass the complete input image into CNN and then take the proposed sub-regions from the output feature maps. These are then passed to the pooling layer followed by fully-connected layer to warp the region into square-features, eventually working into softmax and regressor layer to classify the image and get the bounding box coordinates. Then, the YOLO(You Only Look Once) [19] algorithm was introduced which takes whole input image for predictions instead of the above region-based proposed methods. In this algorithm, the entire image is divided into

a $S \times S$ grid and then m bounding boxes from each grid are selected, which are used to detect the class of an object and predict the offset value of bounding boxes. YOLOv4 [3], a newer version is used as the object detection mechanism in this thesis.

2.3 Latency and Accuracy

Many researchers have centered their work on identifying the deficits or time consuming parts of the perception and proposed different methodologies. RTOD [11], a real-time object detector with minimized end-to-end delay for autonomous driving points out the contentions and delays in object detection and presents zero-slack and contention-free pipeline mechanisms. Although the results show a good improvement in inference rates, the authors do not consider dividing the workloads or implementation on different architectures for comparison. In another paper - DNN-SAM [12], the authors propose a dynamic Split-and-Merge Deep Neural Network execution and scheduling framework that splits the input image into mandatory and optional sub-tasks and processes them individually based on the criticality, merging the results later. Expensive sensors like LiDAR and fusion cameras are used for critical region identification, resulting in good accuracy but high latency. Also, since a scheduler is designed to run the tasks, it is not guaranteed that the optional sub-tasks might always be scheduled due to less or no available time slice. Apart from these, there has been study to improve computation of image data and neural networks on GPU. [17] studies the effect of co-scheduling multiple image processing tasks on the GPU and improve it. Similarly, [26] presents a system solution that optimizes the execution of DNN workloads on GPU in a real-time multi-tasking environment. These papers focus

on the conflicts and issues while using GPU but not on type of task in consideration or an application specific usage. In our work, we focus on obtaining an optimal balance of accuracy and latency by utilising all the hardware resources and dividing workloads into parallelizable tasks.

Chapter 3

System Design

This chapter focuses on the implementation details and the design choices. We first introduce the overall system design, followed by the perception methodologies used, and then present a runtime mechanism to execute individual perception workloads of the system on CPU and GPU resources based on their criticality and execution time. Along with this, we propose an approach to determine the critical region in the frame.

3.1 Overall Design

This section describes the overall design of our system. Autonomous driving as an application has stringent timing constraints and has to be accurate in analyzing the surroundings. In this thesis, we present an approach to reducing the end-to-end delay of analyzing the scene for an autonomous driving application without compromising on the accuracy. We develop a small-scale prototype to implement and test our approach which can easily be ported to other platforms and also be subjected to real-world environments.

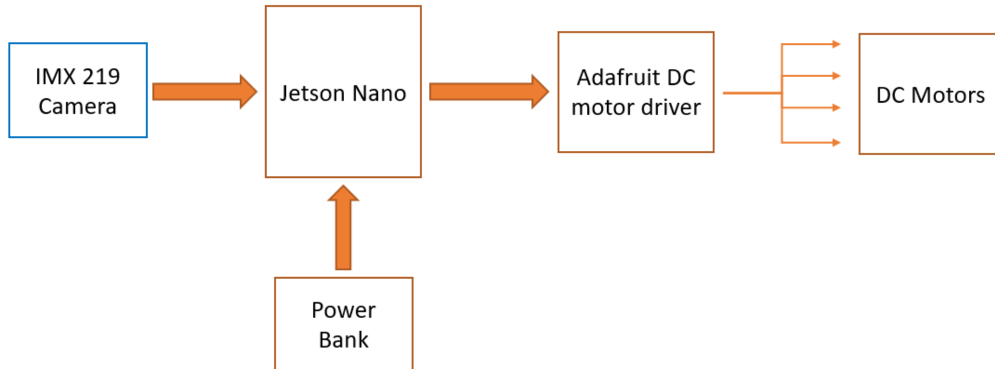

Figure 3.1: Hardware Block Diagram

The small-scale prototype consists of a computing platform, a single camera sensor mounted in the front of the car, a motor driver to drive the four DC motors on each wheel and a power bank to power the main board. Fig 3.1 shows the hardware block diagram of our implementation. All the hardware components can be spotted. In order to choose each of these, we take inspiration from NVIDIA's Jetbot [16] and use the following components for implementation:

- **Jetson Nano**: This is the computing platform used. It has 128-core Maxwell architecture GPU, quad-core ARM CPU, 4GB 64-bit LPDDR4 RAM and can process videos of up to 4k resolution at 30fps. This makes it both cost and performance efficient. For compute-heavy applications like autonomous driving, the GPU capabilities become very useful.

- **IMX219 Camera sensor**: This camera has a wide angled view that covers 160°of the field. As our work involves the usage of a single camera, we utilize this to capture as much information as possible and process it conveniently on the Jetson Nano.

- **Adafruit DC Motor + Stepper FeatherWing**: This motor driver can control four DC motors or two Stepper motors at once which fits in well for the purpose. It also comes with the Adafruit Motor library for ease of use.

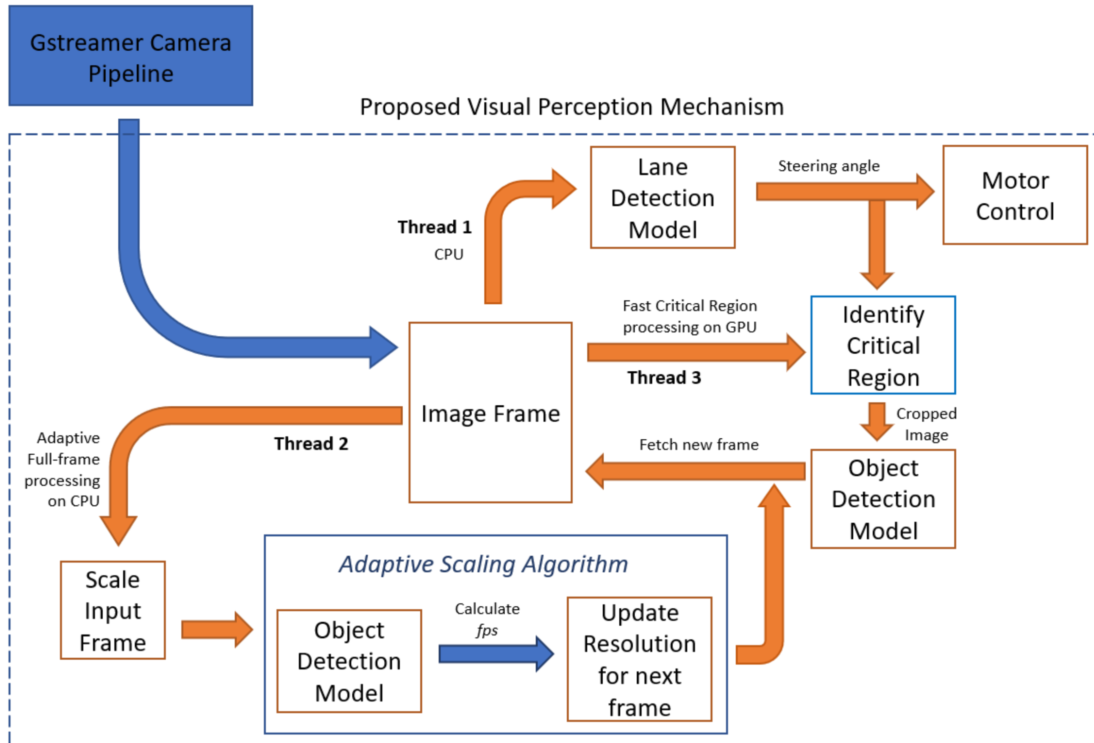

Figure 3.2: Overview of System Flow

Next, we move on to the system design. Fig 3.2 shows the block diagram of the system flow. The entire flow focuses on the visual perception of input camera frames and is divided into three parts handled by three different threads running in parallel. The block 'GStreamer Camera Pipeline' is responsible for continuously updating the frame object 'Image Frame' shared by all the threads. Each of the threads is responsible for a task that helps in understanding the scene. The first thread labeled 'Thread 1' takes the input frame

and runs the lane detection model, giving the steering angle as the output. This output is utilized for controlling the motors as well as by 'Thread 3' for identifying the critical region in the frame. The second thread, labeled as 'Thread 2' in the block diagram takes the input image and runs the object detection model on the entire frame at a higher resolution initially. Based on the time taken for this inference, the resolution of the image fed to the model is adjusted. This is called the 'Adaptive Scaling Algorithm'. It automatically finds the optimal resolution that achieves the 'Target FPS' (Processing rate that does not compromise on latency, yet providing accurate results). Once the resolution is settled, this thread keeps on inferencing the object detection model continuously. The third thread, labeled 'Thread 3' runs another instance of the object detection model, but on the critical region of the image. All these threads keep running simultaneously providing an accurate and fast perception system for the autonomous driving application.

It can also be noticed that the threads 1 and 2 run on CPU but the thread 3 utilizes the GPU resources. The next sections of this chapter explain these design choices and the reason behind them in much more detail.

3.2 Perception Algorithms

Perception - understanding and interpreting the scene, is one of the most important parts in autonomous vehicles. Modern systems use a variety of sensors like RADAR and LiDAR to accurately represent the scene. Information such as depth, distance and direction can be easily provided by such sensors but it does not replicate human vision, meaning it fails to interpret the type of object, the sign and signals important to drive safely in most

scenarios. For such reasons, camera sensors are paired with them or used as a stand-alone as it provides much richer information of the scene.

A camera sensor can help in detecting a lot of information including traffic light identification and classification, lane detection, and last and the most important - for object detection and classification. As mentioned earlier, in our implementation, only a single camera is used to capture the frames from the front of the car. Since our work focuses on optimizing the visual perception by utilizing the available resources, a single wide-angled camera serves the purpose. The wide angle of 160° on IMX219 helps in gathering as much as possible information from the scene.

In order to analyse the scene, we need to identify the lane and objects in the frame. Since the implementation is a small-scale working model, it is important for the car to stay on track while analyzing the objects on the way. Therefore, we utilise the lane and object detection mechanisms for our approach.

3.2.1 Lane Detection

For lane Detection, we use the DAVE-2 model as suggested before, shown in Fig 3.3. The weights of the network are trained to minimize the mean-squared error between the steering output by the network and the ground truth (actual angle of rotation required to stay in the lane). Since the model is trained on the video stream from the dashboard of the car to output the steering angle, it fits in very well in the implementation of a working model for the prototype. The output steering angle is restricted between -30° and 30° as it is sufficient for accuracy and performance, allowing movement in all directions.

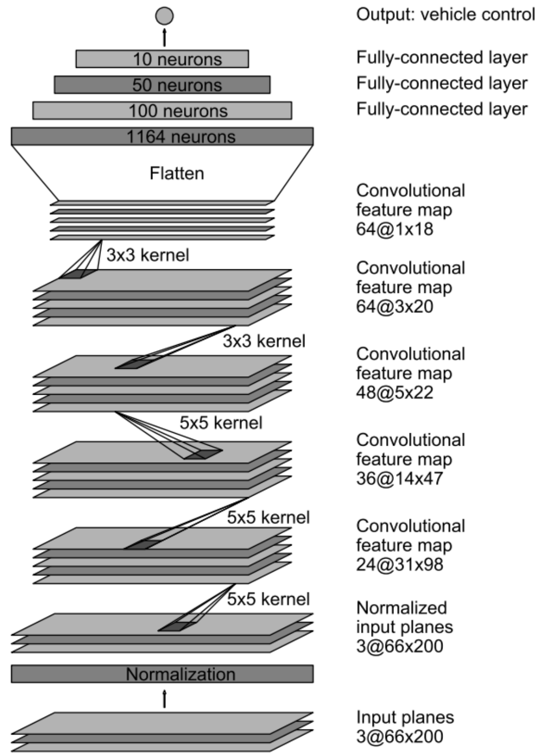

Figure 3.3: CNN Architecture of DAVE-2 Model [4]

Now in order to imitate and make the lane detection model work for our use case (on small scale), we need the dataset that suits. As the dataset used in [4] is from real-world and we need to make the car to move on a track taped on the floor, we retrain the model using the dataset used and provided by the authors in [1]. Retraining the DAVE-2 model using the suggested hyper-parameters, epochs and batch size in [1], we obtain an accurate way of detecting the lane.

3.2.2 Object Detection

Next we move on to the next section of perception - the object detection. Since the advancement of convolution neural networks (CNN), massive research has been done in the field of object detection which has shown many practical applications in autonomous vehicles, robots, and much more. Various deep-learning methods have been proposed since then like R-CNN, fast R-CNN, faster R-CNN and You Only Look Once (YOLO). The reason for YOLO to be widespread and used in many real-world applications is for its accurate and high inference computation time compared to above deep learning based object detection methods.

Different versions of YOLO - The first version of YOLO (YOLOv1), performed detections without processing the image multiple times hence justifying its name. They divide the input image into predefined multiple grids ($S \times S$) and if the center of an object falls into that grid, it is then responsible for the detection. Each grid cell then predicts the class probabilities and the bounding boxes, B. For each of the bounding boxes, the model outputs four coordinates - x, y (center coordinates of the box) and h, w (height and width of the box) along with confidence score C. Hence total predictions, $pred = S \times S \times (B * 5 + C)$ for a single input image. The architecture for this version of YOLO has 24 convolutional layer and 2 dense prediction layer and the Tiny-YOLO version contains only 9 convolutional layer. The major problem of this version is if the grid contains multiple objects then the model will not predict all of them.

After various versions of YOLO, the recent version YOLOv4 [3] is used in our method. This paper introduces various new modules which increase the performance of ob-

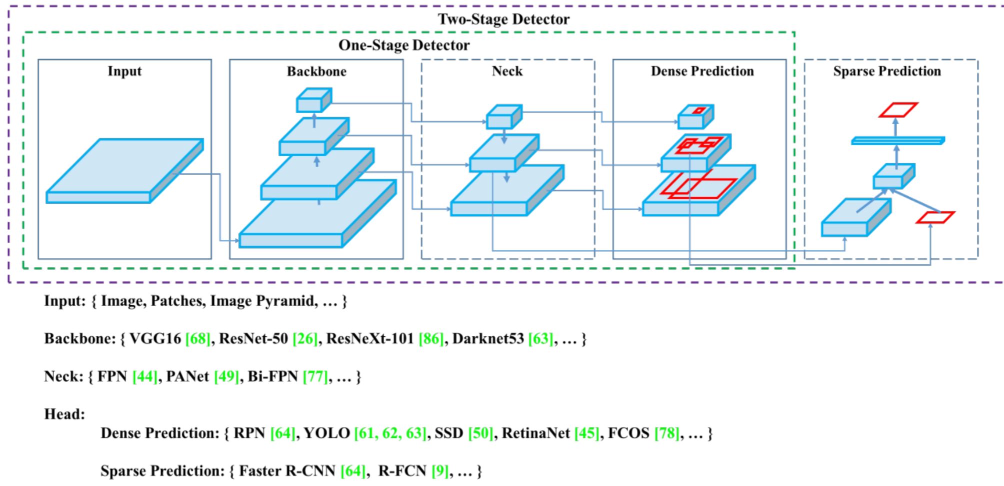

Figure 3.4: Overview of YOLOv4 Object Detector [3]

ject detection like weighted-residual-connections (WRC), Cross-Stage-Partial-connections (CSP), Cross mini-Batch Normalization (CmBN), Self-adversarial-training (SAT) and Mish-activation. It also adopts the dropblock regularization block and mosiac data-augmentation mechanism. Figure 3.4 shows the complete architecture of YOLOv4 where the input image is passed into the backbone (RESNET-based backbone used for our method) which is further passed into the Neck module (contains FPN/Bi-FPN block) and finally given to the YOLO dense prediction module. The authors of YOLOv4 have shown the impact different resolution images has on the *fps* and accuracy. They show that for the same backbone architecture, the *fps* reduces and accuracy increases as the resolution of the image is increased. We utilize this property of YOLO throughout our proposed system.

3.3 Critical Region Identification

Autonomous driving as an application is very critical as it requires an appropriate level of control and accuracy while understanding and evaluating the scene. Due to that, identifying the critical parts of the frame - for example, calculating the time left to avoid a potential collision with some object right in front of the car or recognising both nearer as well as farther objects in the frame, becomes an essential part of the process.

A lot of computation and memory resources go into this. Most of the implementations use expensive sensors like LiDAR and RADARs or computationally expensive depth estimation algorithms to realise the critical part of the scene. For our implementation though, we utilize the existing resources to come up with a simple yet efficient way. We define the critical region as the area right in front of the car, the region that covers most of the relevant information of the scene. Since we have a lane detection model that identifies the lane and provides the steering angle as the output, we already have the information needed to find the critical region. In other words, we utilize the angle to identify the region in front of the car (critical region). For example, if the lane detection model gives out the steering angle as 30°, then the critical region is identified as the bottom right part of the frame as the car is now set to turn right. An example is shown in Fig 3.4. Since the lane detection model outputs angle between -30° and 30° the critical region is appropriately identified for each and every angle in between which makes it more precise.

Once the critical region is identified, it is cropped and processed separately as explained in the later sections of this chapter. There are multiple reasons for choosing the proposed way of identifying the critical section:

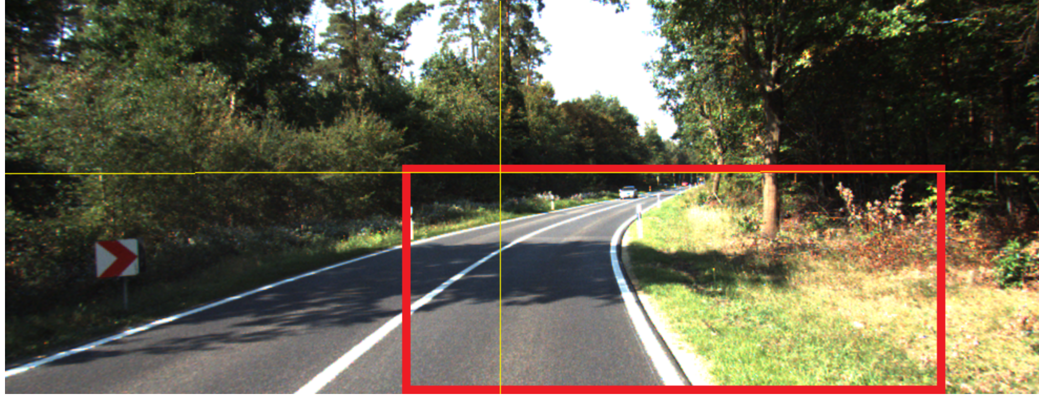

Figure 3.5: Critial Region Identified for Steering Angle of 20°. [6]

- The region right in front of the car changes at a faster rate and may contain smaller objects (also the farther ones) that the might be difficult to identify in the entire frame.

- The Jetson Nano platform being resource constrained, we need to be decisive on what to utilize the resources on. Since we also run the YOLOv4 object detection model on the cropped region, we cannot afford to spend more time on identifying it.

- The output from the lane detection model is very well utilized for this purpose. This reduces the unnecessary processing time needed for any other approach, thus giving a high throughput.

3.4 Runtime Mechanism with Adaptive Scaling

The previous sections mentioned about the perception algorithms used in our system and the proposed critical region identification method. In this section, we propose a runtime mechanism that constructs perception into multi-threaded tasks and executes

them in parallel in order to achieve better performance. For the sake of understanding the scene accurately and in a timely manner, we divide the entire perception workload into the following tasks:

- The first task is to inference the lane detection model and obtain the steering angles to drive the car as well as identify the critical region.
- The second task is to inference the object detection model on the entire input frame and identify the bigger objects in the frames captured by the wide-angled camera.
- The third task is to inference the same object detection model on the critical region, thus identifying the smaller objects right in front of the car.

We can challenge the reasoning behind dividing the tasks as above by considering the alternative approaches. Firstly, since the lane detection model is independent of the YOLOv4, it can run in parallel. Next, let us consider that the system just focuses on inferencing the object detection on the entire frame at the best resolution possible without compromising on the throughput i.e. the *fps*. Though the Jetson Nano platform has good computing capabilities, the YOLOv4 being compute heavy still poses some strain on it. And as mentioned in [3], in order to decrease the inference time, we need to reduce the resolution of the input frame. But by doing that, we fail to recognise some of the smaller objects that might be in the critical region of the frame. Hence, we drop this approach.

On the contrary, if the system is designed such that it only inferences the object detection on the identified critical region, we fail to analyse the entire scene. In this case, any potential hazard incoming from outside the frame remains unrecognised until it arrives in the critical region and by the time it is recognised, it might be too late to prevent a

potential accident. As a consequence, we do not opt for this approach as well. Instead, we decide to run both the tasks in two separate threads. These threads do not compete for any shared resource and can run independently. By dividing the tasks, we do not miss on analysing any essential part and rather get a deeper understanding of the scene.

Focusing on the implementation of the proposed approach, we first identify the dependency of the threads on each other. All the thread functions need the most recently captured frame as their input. In order to fulfil this, we assign a global variable accessible to all the threads which keeps updating regularly. Also, to avoid any write while read, we maintain two variables that update alternatively and point the global variable to it. Since the third thread identifies the critical region based on the output of the lane detection model, we declare a shared variable that updates the same way as the frame. For all this implementation, we use the *POSIX thread (pthread)* library due to its effectiveness in achieving parallel and distributed processing.

Adaptive Scaling Algorithm. As mentioned in 3.1, there is an added mechanism that provides for the adaptability of the perception methodology - ‘Adaptive Scaling Algorithm’. This algorithm finds the best resolution that the image can be processed at without compromising on the latency. The algorithm starts with a high resolution (set at 512×512) and calculates the average frames per second (*fps*) achievable at that resolution and keeps changing it until an optimal *fps* (set at 1 fps) is attained. Due to this added functionality, the entire system can be ported to a different computing platform. If a better platform is used, then the target *fps* can also be increased for a faster processing rate and the algorithm can do the rest, settling for the resolution that works best and fully utilizes the computing

capabilities. This adaptability is possible due to YOLOv4's property that it takes less time to process a lower resolution image and vice versa.

3.5 CPU and GPU workloads

The Jetson Nano has 128 core GPU and multi-core CPU. Due to its GPU capabilities, the device provides for a faster inference of CNN models. As seen in the previous section, we divided the system into different tasks. It is now an important factor to decide which task runs on what processing unit as that decides the overall performance of the system. Carefully analysing the tasks, we propose the following:

- The first task (inferencing the lane detection model) is made to utilize the CPU.
- The second task is running the object detection model on the entire frame. We assign this task to another CPU core.
- The last task is inferencing object detection on the critical region, which we assign to the GPU.

In our proposed methodology, we make sure that the third task utilizes the GPU completely with the other two tasks running independently on the CPUs. In section 3.4, we mentioned that the critical region covers most of the scene as it focuses on the region right in front of the car. This means to say that it is important to process the critical region at a faster rate and a higher resolution as compared to the entire scene as any potential hazard is most likely to be caused in this part of the scene. Assigning all the GPU resources to process the critical region ensures this. On the contrary, the entire scene is just a repetition

of the critical region with some extra peripheral vision. Thus, it is fair to process it on the CPU with the highest resolution possible without compromising the processing rate. Designing the system this way ensures that both the farther as well as the closer objects in the frame are recognised as soon as possible.

It can be noticed that with this approach, the smaller objects in the peripheral regions of the scene are not recognised as the resolution at which the entire frame is processed is insufficient (Due to the constraint on the processing rate and limited computing capability on the CPU). But in order to resolve it, if we were to assign both the tasks two and three to the GPU, they would share the resources which will reduce the overall processing rate of the scene and eventually lead to higher latency. Thus, we make a trade off, rather a good one as most of the scene except the critical region is low priority most of the time.

Chapter 4

Evaluation and Results

In this chapter, we evaluate our proposed model using real world dataset, compare it against the other approaches and present the results. We first mention the inference timings of the lane and object detection model on the Jetson Nano at different resolutions. Then we go ahead and select a dataset to conduct a case study using an evaluation criteria. Lastly, we compare the results obtained with different approaches and compare it with our proposal.

4.1 Prototype Implementation and Testing

This section presents the hardware implementation of the small-scale prototype used to test our proposal and compare it with the other approaches. Fig 4.1 shows the prototype of our implementation. All the hardware components shown in Fig 3.1 can be seen here. Once the car is deployed as shown, we then test the lane detection model using a taped track on floor as shown in Fig 4.2. The track is built similar to the one used for

training the lane detection model in [1]. Testing the car on this track, we obtain good results i.e. the car stays in the lane at all times. Realising that the lane detection model works well, this forms the foundation for the next steps of evaluation done in the rest of this chapter.

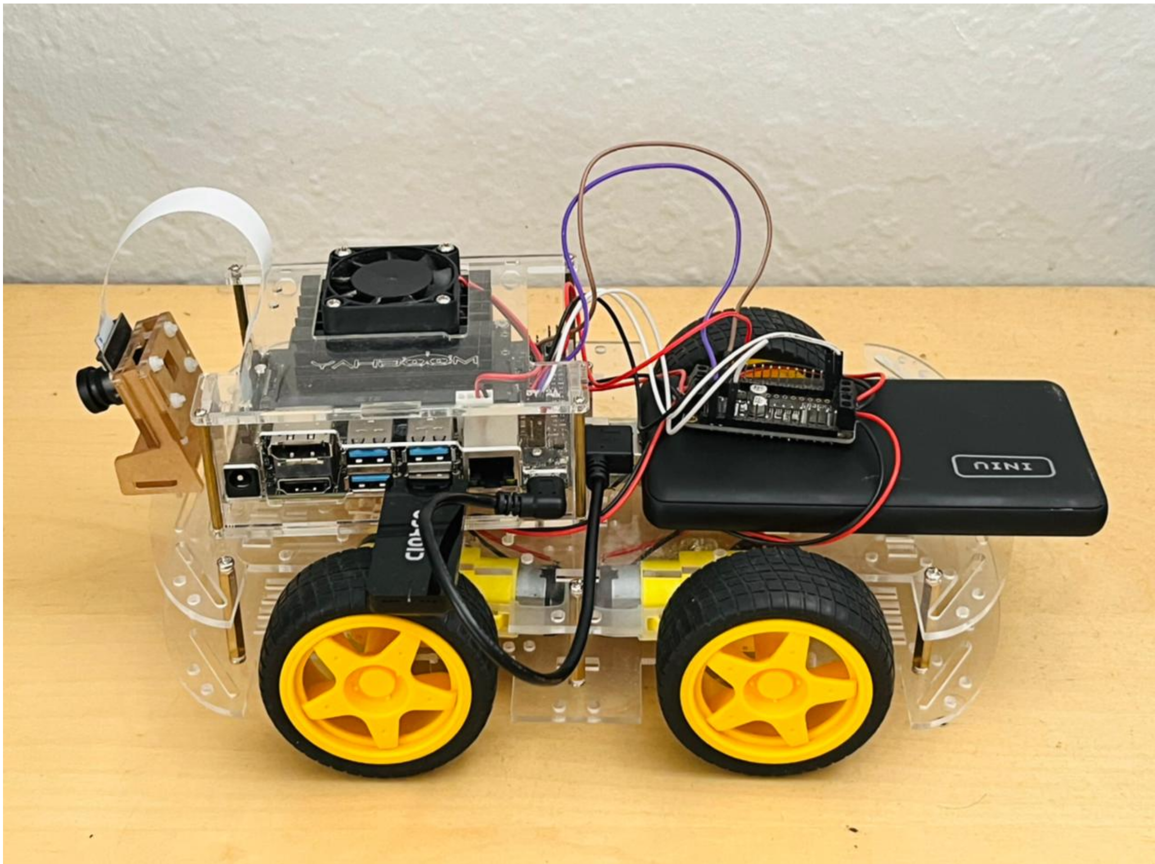

Figure 4.1: Prototype of the Car

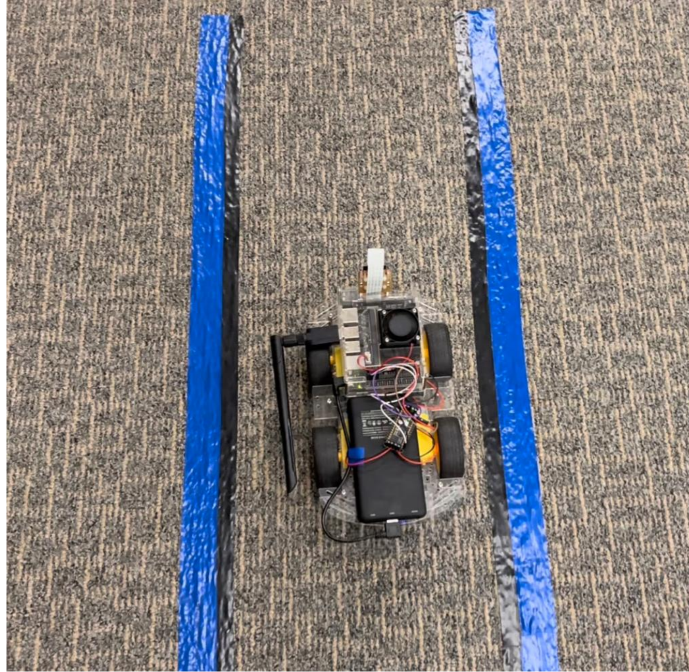

Figure 4.2: Testing the Car on Track

4.2 Inference Timings

As mentioned before, we know that the Jetson Nano has both GPU and CPU cores for computation. We run the program that fetches the image frames on demand from the Gstreamer camera pipeline and feeds it to the lane as well as object detection (YOLOv4) models. We then run it on different resolutions of input image and note the time taken and processing rate. Once we run this on the CPU, we repeat on the GPU to decide the resolution and the corresponding *fps* to be used for our implementation. Table 4.1 shows the results for the YOLOv4. The lane detection (DAVE-2 model) takes same time i.e. 2 *ms* for all resolutions of an input frame. This inference time is very less compared to the inference time for the YOLOv4, which makes it easy to be computed on the CPU.

Image Resolution	CPU		GPU	
	Time taken [ms]	Processing rate [*fps*]	Time taken [ms]	Processing rate [*fps*]
1024×1024	29000	0.03	1480	0.67
512×512	7200	0.14	400	2.5
256×256	1800	0.55	140	7.14
128×128	650	1.5	100	10
64×64	250	4	90	11.11

Table 4.1: Time and processing rates for YOLOv4

Looking at the values above, we see that the time taken by the YOLOv4 is directly proportional to the resolution of the input image and inversely proportional to the processing rate. We also see that the time taken to process a 1024×1024 resolution image on the CPU is way too long. Hence, for the Adaptive Scaling Algorithm presented in the previous chapter, we set the initial resolution to the next lower value i.e. 512×512. Similarly, it can also be noted that the maximum processing rate in *fps* attained by the object detection on CPU is 1.5 *fps* at a resolution of 128×128. If we go lower than this, we heavily compromise on the accuracy of the results as 64×64 is a very low resolution for the objects to be detected. Thus, while choosing the 'Target fps' in the Adaptive Scaling Algorithm, we settle for 1 *fps* as a lower processing speed than this can cause an undesirable delay and potential accident. It must also be noted that these values are for a low-power computing platform like the Jetson Nano. This can be easily scaled up to a better platform like Jetson TX2 which has powerful computation resources - 256 core Pascal GPU architecture with 256 CUDA cores and 8GB of 128-bit LPDDR4 RAM.

Another choice we make is deciding the resolution at which the critical region is inferenced on the GPU. Looking at the table 4.1 above, we see that the processing rate at

128×128 is 7.14 *fps* whereas at 256×256, it is 10 *fps*. Since the difference in the processing rate is less compared to the low accuracy that we will face by using the lower resolution, we settle for the 256×256 resolution when running the object detection on the critical region of the scene.

4.3 Case Study

This section presents the case study conducted to compare our proposed model to other similar approaches.

4.3.1 Dataset Selection

In order to perform this case study, we utilise a real-world autonomous driving dataset - the KITTI Vision benchmark Suite [6] developed by Karlsruhe Institute of Technology and Toyota Technological Institute. This is one of the most popular datasets for use in mobile robotics and autonomous driving application. The dataset consists of hours of videos recorded with different sensor modalities, including high-resolution RGB, grayscale stereo cameras and 3D laser scanners. The full benchmark suite consists of many tasks like stereo, optical flow, visual odometry, 3D object detection and 3D tracking. The dataset is captured and synchronised at 10 *fps*.

For this case study, we use the raw dataset from the suite. There are multiple categories in raw dataset as well that include city, residential, road, campus, person and calibration. We choose a video from the road category because of the simplicity. The video consists of 435 frames stored as individual images with each having a resolution of 1242×375

pixels. There a total of 9 Cars and 1 cyclist approaching towards the autonomous car from far away and passing by, throughout the video. As the dataset is originally recorded at 10 *fps*, we maintain that and fetch the image frames at the same rate, replacing the GStreamer camera pipeline that was used before.

4.3.2 Evaluation Criteria

Having selected the dataset to be used for the case study, we set up an evaluation criteria that forms the basis of the study. As mentioned, the video consists of multiple cars that arrive in the scene at different frames. We define frame in which the car becomes visible and evident to the human eye as **'Frame of Sight'**. If this frame is processed for object detection using YOLOv4 at a high resolution, it must be able to detect the car. Next, we define the frame in which the first object detection results are obtained as the **'Frame of Detection'**, meaning the earliest detection of the object of interest using any of the perception approaches mentioned next. Once we feed the video, we note down the Frame of Detection and compare it with the Frame of Sight. This way, we can calculate the latency with which the object was detected. If there are multiple objects in the frame, then the accuracy can also be determined. Once evaluated, we compare the results with different perception approaches explained next.

Our approach consists of running the lane detection model along with two instances of the object detection model (one on CPU and other on GPU). To challenge this approach, we choose some other approaches that may or may not arguably be as good as our proposed methodology. The other methods used for comparison are:

- For the first approach, we run the YOLOv4 model on GPU at a low resolution (128×128) along with the lane detection model. This approach can process images at a faster rate due to the low resolution but will fail to recognise the smaller objects in the frame i.e. objects that are farther.

- For the second approach, we run the YOLOv4 model on GPU as well but at a higher resolution (512×512) along with the object detection model. This approach can easily recognise the farther objects i.e. at an early stage but will drop a lot of frames in between due to the high processing times.

- This approach is very similar to our proposal. The only difference is that both the YOLOv4 inferences i.e. the entire frame as well as the critical region is processed on the GPU with the lane detection model on the CPU. In this approach, due to shared GPU resources, there is a increase in the processing time, reducing the processing rate.

- The last is our approach.

4.3.3 Evaluation and Results

Before we start evaluation, we select a clip from our selected video dataset. As specified before, there are a total of 435 frames in the video clip. For this case study, we choose a clip where two cars approaching back to back appear. These two cars become visible (to the human eye) at **frame 250** as shown in Fig 4.3. This frame is **'Frame of Sight'**. Any detections that happen in the approaches mentioned above are compared to this frame to arrive at a conclusion. In addition to this, we randomly choose the starting

frame of the clip as **frame 231**. This starting frame number might not seem of much relevance but does matters as it decides the offset when some frames are dropped due to the processing times being higher than the frame rate of the clip in some approaches. But since this is a case study, we randomly choose the starting frame number and proceed with the evaluation. The results do not get affected much and the conclusion remains same as seen next because of different processing rates and latencies.

Figure 4.3: Frame of Sight (frame 250). [6]

We now feed the clip to the different approaches discussed in the previous subsection. The results for each of the approaches is as follows:

- In the first approach (Fig. 4.4), the clip is processed for almost every frame as the processing time ≈100 *ms* matches the input frame rate of 10 *fps*. Thus, there are no frames dropped. This is good but the low resolution makes it hard for the cars to be recognised early. The Frame of Detection for this case is at frame 268, where both the cars are recognised. The frame number input is frame 267. Detection time in recognising the cars is a lot in this approach, found to be ≈1800 *ms*.

Figure 4.4: Input Frame for First Detection in Approach 1. [6]

- In the second approach (Fig. 4.5), the clip is processed every ≈430 *ms*, meaning once a frame is input, it takes ≈430 *ms* to process it. During the time this input frame is processed and the objects in the frame are recognised, approximately **three** (430 *ms*/100 *ms* - 100 *ms*) frames are dropped. This is the drawback of this approach. On the positive side, this approach takes frame 253 as the input and recognises both the cars at frame 257, which is way earlier than the previous approach, reducing the detection time to ≈700 *ms*. Due to the high resolution, the latency remains high but is still significantly reduced compared to previous approach.

Figure 4.5: Input Frame for First Detection in Approach 2. [6]

- The third approach (Fig. 4.6) involves processing both the instances of YOLOv4 on the GPU. Due to this, the resources are shared which leads to the drop in the processing rates of both the full frame as well as the critical region to ≈5 *fps*. Comparing this to the frame rate of the clip (10 *fps*), there is a drop of every alternate frame. In this approach, the cars are recognised at frame 254 where the input frame number was 252, further reducing the detection time to ≈400 *ms*. Due to decent resolution, the latency is reduced compared to the previous approach.

Figure 4.6: Input Frame for First Detection in Approach 3. [6]

- Our approach (Fig. 4.7) dedicates all the GPU resources to the processing of the critical region and using CPU for processing the full frame and running the lane detection model. The critical region processes at a rate of 7 *fps*, meaning it drops a frame for every 3 input frames. This is better than the previous approaches. Both the cars appearing at frame 250 are recognised at frame 253, where the input frame is 252. Here, the detection latency is the least i.e. ≈300 *ms*. This result is better than all the previous approaches in terms of latency, proving to be the best method for fast and accurate perception in autonomous driving application.

Figure 4.7: Input Frame for First Detection in Approach 4. [6]

Figure 4.4 - 4.7 shows the input frames in which the desired two cars are first detected for each of the approaches mentioned. Table 4.2 summarizes the results for a quick comparison. In all the approaches, the desired accuracy is achieved as both the objects are recognised as cars. On the other hand, the latency reduces from approach 1 through 4 with our work taking the least amount of time to detect the cars.

Approach taken	Input Frame	Frame of Detection	Processing Rate [fps]	Detection Latency [ms]
1	267	268	10	$\approx$ 1800
2	253	257	2.5	$\approx$ 700
3	252	254	5.1	$\approx$ 400
4 (Our Work)	252	253	7	$\approx$ 300

Table 4.2: Results for the Case Study

Chapter 5

Conclusions

Real-time perception for autonomous driving has evolved a lot over the past few years and employs various aspects of understanding the scene, including but not limited to lane detection, object detection, depth estimation, critical region identification etc, for which often expensive setups are used. In this thesis, we propose a fast and accurate visual perception of the scene that uses lane and object detection. By utilising a single camera input, we inference only convolutional neural networks for scene processing and present an approach that fully utilises the hardware resources to get an optimal balance of accuracy and latency. We also present an adaptive scaling algorithm that makes the system hardware independent and can be deployed on better or worse computing platforms.

In the case study, we make use of real-world dataset to compare our approach with similar other variations. Conducting the study, we see that our proposed model detects the objects in the scene the earliest and with good accuracy.

5.1 Future Work

As of the current work and implementation, we only focus on the lane and object detection aspects of visual perception in autonomous driving. But more robust features like depth estimation are also needed for real-world scenarios. For that, two cameras can be used instead of one (utilising the second camera port on Jetson Nano) for stereo vision as there are multiple fast and accurate depth estimation techniques using stereo cameras. The critical region identification can then be done based on outputs from both the lane as well as depth estimation. Further, since the system is scalable, it can be ported to different platforms and compared for real-time operation. Also, with added functionality, further division of tasks based on their inter-dependency as well as assigning them to CPU/GPU for optimal accuracy and latency can be explored.

Bibliography

[1] Michael Garrett Bechtel, Elise McEllhiney, and Heechul Yun. Deeppicar: A low-cost deep neural network-based autonomous car. *CoRR*, abs/1712.08644, 2017.

[2] Luca Belluardo, Andrea Stevanato, Daniel Casini, Giorgiomaria Cicero, Alessandro Biondi, and Giorgio Buttazzo. A multi-domain software architecture for safe and secure autonomous driving. In *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 73–82. IEEE, 2021.

[3] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[5] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[6] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[7] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[8] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

[9] M Himmelsbach, A Müller, Thorsten Luettel, and Hans-Joachim Wuensche. Lidar-based 3d object perception. 10 2008.

[10] Yu-Shun Hsiao, Siva Kumar Sastry Hari, Michał Filipiuk, Timothy Tsai, Michael B Sullivan, Vijay Janapa Reddi, Vasu Singh, and Stephen W Keckler. Zhuyi: Perception processing rate estimation for safety in autonomous vehicles. *arXiv preprint arXiv:2205.03347*, 2022.

[11] Wonseok Jang, Hansaem Jeong, Kyungtae Kang, Nikil D. Dutt, and Jong-Chan Kim. R-TOD: real-time object detector with minimized end-to-end delay for autonomous driving. *CoRR*, abs/2011.06372, 2020.

[12] Woosung Kang, Siwoo Chung, Jeremy Yuhyun Kim, Youngmoon Lee, Kilho Lee, Jinkyu Lee, Kang G. Shin, and Hoon Sung Chwa. Dnn-sam: Split-and-merge dnn execution for real-time object detection. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 160–172, 2022.

[13] Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Proceedings of the ACM/IEEE 4th international conference on cyber-physical systems*, pages 31–40, 2013.

[14] Xinran Li, Kuo-Yi Lin, Min Meng, Xiuxian Li, Li Li, and Yiguang Hong. Composition and application of current advanced driving assistance system: A review. *CoRR*, abs/2105.12348, 2021.

[15] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason 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*, pages 751–766, 2018.

[16] NVIDIA. Jetbot.

[17] Nathan Otterness, Vance Miller, Ming Yang, James H. Anderson, F. Donelson Smith, and Shige Wang. Gpu sharing for image processing in embedded real-time systems . 2016.

[18] Dean A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *NIPS*, 1988.

[19] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[20] Hsin-Hsuan Sung, Yuanchao Xu, Jiexiong Guan, Wei Niu, Shaoshan Liu, Bin Ren, Yanzhi Wang, and Xipeng Shen. Enabling level-4 autonomous driving on a single 1 off-the-shelf card. *CoRR*, abs/2110.06373, 2021.

[21] Wei Wang, Hui Lin, and Junshu Wang. Cnn based lane detection with instance segmentation in edge-cloud computing. *Journal of Cloud Computing*, 9(1):27, May 2020.

[22] Yidi Wang, Mohsen Karimi, Yecheng Xiang, and Hyoseung Kim. Balancing energy efficiency and real-time performance in GPU scheduling. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 110–122. IEEE, 2021.

[23] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

[24] Yecheng Xiang, Yidi Wang, Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim. AegisDNN: Dependable and timely execution of DNN tasks with SGX. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 68–81. IEEE, 2021.

[25] Hengyu Zhao, Yubo Zhang, Pingfan Meng, Hui Shi, Li Erran Li, Tiancheng Lou, and Jishen Zhao. Driving scenario perception-aware computing system design in autonomous vehicles. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 88–95, 2020.

[26] Husheng Zhou, Soroush Bateni, and Cong Liu. S^3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 190–201, 2018.

[27] Qi Zhu, Wenchao Li, Hyoseung Kim, Yecheng Xiang, Kacper Wardega, Zhilu Wang, Yixuan Wang, Hengyi Liang, Chao Huang, Jiameng Fan, et al. Know the unknowns: Addressing disturbances and uncertainties in autonomous systems. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.