

# A Decentralized Vision-based Perception System using GPUs and High Definition Map Features

Manato Hirabayashi



# Abstract

Autonomous driving systems have garnered extensive attention as a promising solution to various problems associated with transportation. To realize *high-level* autonomous driving systems, large volumes of data must be processed from external sensors with minimal delays to semantically perceive the surroundings and circumstances of ego vehicles. Tasks that involve data parallelism are often managed using graphics processing units (GPUs) to accelerate computation using a technique known as “general-purpose computing on GPUs.” However, whether a task can be accelerated using GPUs is case specific. Moreover, in the actual deployment of on-board autonomous driving systems, one must consider not only delays but also concerns such as precision and severe power limitations.

The primary objective of this study is the exploration of methods to accelerate perception tasks while fulfilling the criteria required for autonomous driving systems. Hence, the validity of GPUs as acceleration units for the perception tasks of autonomous driving systems are confirmed using two concrete examples. Subsequently, some concerns regarding the application of GPUs to on-board autonomous driving systems are considered, and a possible solution model is proposed. Thereafter, a performance analysis of a prototype model is presented to clarify the benefits of decentralized processing, followed by a summary of the conclusions. In this study, three concrete topics are addressed regarding perception tasks that use GPUs: (1) traditional image-based object detection, (2) traffic-light-state recognition, and (3) presentation of a decentralized processing model as well as performance analysis of its prototype.

After providing an introduction and a discussion of related studies, the application of GPUs for accelerating traditional pattern recognition tasks

---

is presented herein. In this study, traditional image-based object detection tasks are considered. Because advanced driver-assistance systems, which are widely implemented in many commercial vehicles, are typically based on traditional pattern recognition techniques, the approaches for these traditional techniques are expected to provide insights into the development of autonomous driving systems. A detailed workflow of the object detection algorithms is presented, with emphasis on how each component can be accelerated using GPUs. Subsequently, evaluation results are provided to quantify the performance improvements achieved via GPU-based implementations. Based on a detailed analysis of the workflow, it was discovered that approximately 98 % of the entire computation exhibits properties that have a high affinity for acceleration by GPUs. In the best scenario of GPU implementation, a performance improvement of  $8.6\times$  was achieved compared with a high-end central processing unit implementation, without changes to the algorithm flow. Subsequently, the current mainstream image-based object detection and traditional methods are compared. Based on the comparison, it is concluded that applying the mainstream approach is inevitable for implementing the perception modules of high-level autonomous driving systems.

Thereafter, a scheme to recognize traffic light states from images is reviewed to verify the practical application of GPUs to perception tasks. Because autonomous self-driving vehicles are expected to share roads with vehicles driven by people during the transition period for their introduction, autonomous vehicles must be able to recognize a wide range of traffic information, such as road signs and traffic lights. The proposed scheme for traffic-light-state recognition comprises two main parts: (i) utilizing ego-vehicle locations on high-definition three-dimensional maps to extract regions of traffic lights from images; (ii) utilizing a deep-learning-based recognizer that requires GPU acceleration. Using practical datasets obtained from public driving experiments, the proposed scheme yielded an average precision exceeding 97 % under favorable conditions. Moreover, if the recognition targets were within a distance of 90 m, then a recognition recall of approximately 90 % was achieved.

Possible concerns that may arise when applying GPUs to on-board

---

autonomous driving systems are also discussed. As a countermeasure to these concerns, a model of a decentralized processing system using embedded-oriented GPUs as decentralized units is presented. To confirm the validity of the proposed model, a prototype is implemented for image-based object detection, and its performance analysis is discussed. The quantitative evaluations show that an approximate delay of 27 ms on average occurred between feeding an image to the system and receiving the object detection results by the host. This indicates that although the measured delay includes overhead from decentralized processing, frame dropping did not occur under the experimental setup conditions. Moreover, the proposed model degraded the network load of the host by several orders of magnitude, indicating its robustness against system scaling. Based on quantitative evaluations, it is concluded that autonomous driving systems can benefit from the proposed decentralized processing scheme, even with delays involved.

# Contents

<b>Abstract</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.1.1 Autonomous driving as an emerging technology . . . . .	1
1.1.2 General-purpose computing on GPUs . . . . .	3
1.2 Contributions and structure of this study . . . . .	7
1.3 Organization . . . . .	11
<b>2 Related studies</b>	<b>12</b>
2.1 Object detection . . . . .	12
2.1.1 Traditional algorithms . . . . .	12
2.1.2 Cutting-edge methods using revived neural network techniques . . . . .	13
2.2 GPUs as decentralized units . . . . .	14
<b>3 GPU-Accelerated Object detection</b>	<b>17</b>
3.1 Accelerating traditional object detection algorithm using GPUs	17
3.1.1 Preliminary analysis of DPM workflow . . . . .	18
3.1.2 GPU-based implementation of DPM . . . . .	21
3.1.3 Results of DPM acceleration using GPUs . . . . .	32
3.1.4 Results of GPU-accelerated traditional object detection	51
3.2 Paradigm shift forming current mainstream . . . . .	52
<b>4 Traffic light recognition using High Definition Map Features</b>	<b>56</b>
4.1 Motivation . . . . .	56

## CONTENTS

---

4.2	Proposed scheme and assumptions . . . . .	57
4.2.1	3D High Definition Maps . . . . .	58
4.2.2	ROI Extraction . . . . .	59
4.2.3	Morphological Processing . . . . .	61
4.2.4	Deep-Learning-Based Detector . . . . .	63
4.3	Implementation . . . . .	66
4.3.1	Autoware Implementation . . . . .	66
4.3.2	Color State Training and Recognition by SSD . . . . .	67
4.3.3	Interframe Filter . . . . .	70
4.4	Evaluation of the proposed scheme using practical data . . . . .	71
4.4.1	Experimental Setup . . . . .	72
4.4.2	Effect of ROI Extraction on Recognition Accuracy . . . . .	72
4.4.3	Effects of Distance from Target on Recognition Recall . . . . .	78
4.4.4	Effects of ROI Extraction on Recognition Speed . . . . .	80
4.4.5	Effects of GPU on SSD Recognition Speed . . . . .	81
4.4.6	Effects of the Number of Images in the Training Data on Recognition Accuracy . . . . .	82
4.4.7	Discussion . . . . .	83
4.5	Interim conclusions on traffic light recognition . . . . .	84
<b>5</b>	<b>Distributed Perception Architecture Using GPUs</b>	<b>86</b>
5.1	Problems that may arise when applying GPUs to autonomous driving systems . . . . .	86
5.2	Possible solution to the problems . . . . .	87
5.3	Model of a decentralized processing system for autonomous driving . . . . .	88
5.4	Implementation details of a prototype of decentralized pro- cessing system . . . . .	91
5.4.1	Robot operating system (ROS) implementation . . . . .	92
5.4.2	Deep-learning inference using TensorRT and weights quantization . . . . .	93
5.4.3	Jetson settings . . . . .	94
5.5	Performance analysis . . . . .	94
5.5.1	Experimental setup . . . . .	94

---

5.5.2	Delays based on decentralized processing . . . . .	95
5.5.3	Effects of weights quantization on resource and power consumptions . . . . .	99
5.5.4	Limitations of centralized processing . . . . .	102
5.5.5	Effects of weights quantization on detection perfor- mances of different series of edge devices . . . . .	105
<b>6</b>	<b>Discussion</b>	<b>110</b>
6.1	GPU-accelerated perception algorithms . . . . .	110
6.2	Decentralized processing using embedded-oriented GPUs . . .	111
<b>7</b>	<b>Conclusion</b>	<b>116</b>
7.1	Summary of contributions . . . . .	116
7.2	Future direction . . . . .	118
	<b>Acknowledgement</b>	<b>120</b>
	<b>Bibliography</b>	<b>131</b>
	<b>List of publications by the author</b>	<b>132</b>



# List of Figures

1.1	Mapping of threads, blocks, and grids in CUDA. . . . .	5
3.1	Overview of the DPM algorithm . . . . .	19
3.2	Breakdown of the execution time of a DPM . . . . .	20
3.3	Concept of loop unrolling. . . . .	22
3.4	Computation of a score array in DPM. . . . .	24
3.5	Loosely unrolled parallelization of a score array. . . . .	25
3.6	Tightly unrolled parallelization of a score array. . . . .	26
3.7	Hybrid parallelization of the score array. . . . .	27
3.8	Approach for constructing a HOG pyramid. . . . .	28
3.9	GPU-based implementation schemes using the bilinear algorithm	29
3.10	Execution times of the DPM program on GPUs and CPUs. . .	35
3.11	Breakdown of the execution times on two platforms. . . . .	37
3.12	Effects of the parallelization schemes on part score computations.	38
3.13	Effects of different parallelization schemes on image resizing. .	39
3.14	Effects of texture memory on part score computations. . . . .	40
3.15	Effects of texture memory on HOG feature computations. . . .	41
3.16	Effects of shared memory on HOG feature computations. . . .	42
3.17	Effects of input image resolution on various GPUs. . . . .	43
3.18	Breakdown of the execution times for different image resolu- tions on the worst- and best-performing GPUs. . . . .	44
3.19	Effects of color and grayscale images. . . . .	45
3.20	Effects of object classes (vehicle and pedestrian). . . . .	46
3.21	Effects of maximum number of threads per block in loosely unrolled parallelization. . . . .	47

---

3.22	Effects of maximum number of threads per block in tightly unrolled parallelization. . . . .	47
3.23	Effects of resizing algorithms. . . . .	48
3.24	Data flow in DPM. . . . .	49
3.25	Effects of memory copies between the host and device memories	49
3.26	Winning scores of the ILSVRC classification tasks . . . . .	53
4.1	Proposed ROI extraction definition. . . . .	62
4.2	ROI extraction overview. . . . .	63
4.3	Flow diagram for morphological recognition. . . . .	64
4.4	Flow of recognition with the SSD approach. . . . .	65
4.5	Overview of data connections between Autoware and the proposed method. . . . .	67
4.6	Breakdown of the training dataset. . . . .	68
4.7	Proportions of the ground truth images in each evaluation dataset. . . . .	73
4.8	Effects of ROI extraction on recognition accuracy. . . . .	73
4.9	Recognition precision and recall by color for each dataset. . . . .	75
4.10	Recognition recall shifts relative to distances from the target traffic lights. . . . .	79
4.11	Effects of ROI extraction on recognition time. . . . .	81
4.12	Effects of GPU on recognition time. . . . .	82
4.13	Recognition accuracy shifts relative to the number of images in the training dataset. . . . .	83
5.1	System model of decentralized processing. . . . .	89
5.2	Connection diagram to evaluate data delays on decentralized processing . . . . .	96
5.3	Delay comparisons with and without decentralized processing. . . . .	98
5.4	GPU computational resource utilization for different operation precisions . . . . .	100
5.5	GPU memory utilization for different operation precisions measured during the experiments in Fig. 5.4. . . . .	101
5.6	Power consumed by the GPU for different operation precisions measured during the experiments in Fig. 5.4. . . . .	102

## LIST OF FIGURES

---

5.7	Output rates of object detection results in the centralized environment. . . . .	103
5.8	Utilization of GPU computational resources on the host for different numbers of tasks. . . . .	104
5.9	Execution time comparisons for power budget and operation precision on the Jetson AGX Xavier. . . . .	107
5.10	Execution time comparisons for power budget and operation precision on the Jetson Nano. . . . .	108

# List of Tables

1.1	Summary of levels of driving automation . . . . .	2
1.2	Execution times for resizing an image using texture memory . . . . .	6
1.3	General characteristics for various types of external sensors leveraged in autonomous driving . . . . .	7
2.1	Comparison between several products from the Jetson series . . . . .	15
3.1	Data that can be stored in the texture memory. . . . .	31
3.2	Key specifications of GPUs used in the evaluations. . . . .	34
3.3	Actual execution times corresponding to Fig. 3.10 (in milliseconds). . . . .	36
3.4	File sizes of model filters for pedestrians and vehicles. . . . .	45
3.5	Performance comparison of detection algorithms . . . . .	54
4.1	Threshold values used in morphological processing . . . . .	64
4.2	Training data examples for traffic light recognition. . . . .	69
4.3	Interframe filter for morphological processing . . . . .	70
4.4	Interframe filter for machine learning recognition method. . . . .	71
4.5	Evaluation dataset. . . . .	72
4.6	Confusion matrices for each recognition method (daytime dataset). . . . .	76
4.7	Confusion matrices for each recognition method (sunset dataset). . . . .	77
5.1	Part of the power-mode presets for the Jetson AGX Xavier . . . . .	91
5.2	Precision and recall comparisons for different operation precisions for six specific categories on the MS COCO val 2014 dataset. . . . .	106

## LIST OF TABLES

---

6.1	Comparison of network traffic amounts with and without decentralized processing under experimental settings. . . . .	112
-----	--	-----



# Chapter 1

## Introduction

### 1.1 Background

#### 1.1.1 Autonomous driving as an emerging technology

Autonomous driving systems have been suggested as a promising solution to transportation problems, including the prevention of serious traffic accidents and personal transportation modes in aging societies. A survey conducted by the Cabinet Office of Japan in 2014 [1] revealed that rapidly aging populations are a global concern; the survey indicated that the percentage of people aged 65 and over increased from 5.1 % in 1950 to 7.7 % in 2010, and the percentage is expected to increase to approximately 17.6 % by 2060. In addition, the survey indicated that this increasing trend is expected to continue in the latter half of the century. Personal transportation is therefore a critical emerging problem in aging societies. People living in areas without effective public transportation systems often rely on driving. However, many older people have difficulty driving owing to the natural decline in their physical and cognitive abilities.

The Society of Automotive Engineers (SAE) International of USA published the “Levels of driving automation” (SAE J3016) [2] guideline in 2016, and various companion guidelines have been developed in other countries; for example, the Society of Automotive Engineers of Japan (JSAE) established the “Taxonomy and definitions for terms related to driving automation

Table 1.1: Summary of levels of driving automation  
(cited and modified from ref. [2])

Level	Name	Dynamic driving task (DDT)		DDT fallback	Operational design domain
		Sustained lateral and longitudinal vehicle motion control	Object and event detection and response		
[Lv. 0–2] Driver performs part or all of the DDT					
0	No Driving Automation	Driver	Driver	Driver	n/a
1	Driver Assistance	Driver and System	Driver	Driver	Limited
2	Partial Driving Automation	System	Driver	Driver	Limited
[Lv. 3–5] System performs the entire DDT (while engaged)					
3	Conditional Driving Automation	System	System	Fallback-ready user	Limited
4	High Driving Automation	System	System	System	Limited
5	Full Driving Automation	System	System	System	Unlimited



## 1.1. Background

---

systems for On-Road Motor Vehicles” [3] in 2018. Table 1.1 shows a summary of the levels of driving automation established by the SAE; as shown, a minimum of level three is typically required for autonomous driving to be a solution to transportation problems in aging societies. According to [2], levels three and above refer to cases wherein “*the automated driving system performs the entire dynamic driving task (DDT) on a sustained basis while engaged*”. In other words, autonomous driving systems with these levels are responsible for continuously performing the entire driving task, including steering, acceleration, and braking. To achieve such highly automated systems, we consider driving tasks as a cycle of three primary elements, i.e., *perception*, *planning*, and *control*, and implement modules such that the elements function cooperatively. Perception tasks consider data captured by external sensors and perform semantic understanding with respect to the volatile surroundings and circumstances of ego vehicles. Subsequently, the results are transferred to the two remaining tasks: planning tasks, which determine the manner by which the ego vehicles should move, and control tasks, which compute and transmit actual operations for movement (e.g., the steering angle and acceleration/deceleration amount) to the ego vehicles. Whereas typical perception tasks are computationally intensive and take large volumes of data (e.g., images from high-resolution cameras or point clouds from LiDARs) as inputs, lengthy processing times are not tolerated because processing delays during the perception tasks can directly cause delays in the entire system, which may affect the safe driving conditions of the system as the subsequent planning and control tasks depend on the outputs of the perception tasks.

### 1.1.2 General-purpose computing on GPUs

Massively parallel computing with the assistance of graphics processing units (GPUs) has recently garnered considerable attention for use in computationally intensive applications. GPUs were initially designed as hardware devices for accelerating graphics processing; however, the significant computing capabilities of GPUs have enabled general-purpose solutions to computing data-parallel workloads. Application of GPUs for computationally intensive

workloads instead of graphics is often referred to as *general-purpose computing on graphics processing units* (GPGPUs). The emergence of GPGPU technology has increased the popularity of GPUs in many scientific fields. In particular, after the remarkable success of deep convolutional neural networks (CNNs) in a competition for recognizing large-scale general scenes [4], GPUs have been regarded as an essential device for supporting recent breakthroughs in deep learning because their computing capabilities have been widely used to train deep neural networks with a large number of parameters.

CUDA [5], which is a C-based integrated programming environment developed by NVIDIA, is one of the most popular programming languages for GPGPUs. Some of the representative features of CUDA are highlighted below.

### **CUDA Threads**

The minimum computational unit of the CUDA is a *thread*. During massively parallel computing, thousands of threads are executed in parallel. The CUDA employs a hierarchical structure to simultaneously control large numbers of threads. A collection of threads is termed a *block*, and a collection of blocks constitutes a *grid*. A grid is generated each time the central processing unit (CPU) of a system issues a command to use the GPU, and the blocks and threads in the grid are mapped in three dimensions. This hierarchical structure of CUDA threads is illustrated in Fig. 1.1.

The CUDA assigns streaming multiprocessors (SMs) as GPU resources to each block. Each SM contains multiple streaming processors (SPs), and each thread is executed on an SP. In each SM, the computations are managed by a unit called a *warp*, which is a collection of multiple threads executed in a group; typically, a warp contains 32 threads. Because only one program counter exists for each warp, if a block contains numerous threads but some of them remain in an idle state owing to conditional branches, etc., then the GPU resources are wasted, and the throughput is reduced. Hence, programmers often combine multithreading on the CPU and CUDA threads on the GPU such that the idle CUDA threads on the GPU are hidden maximally.

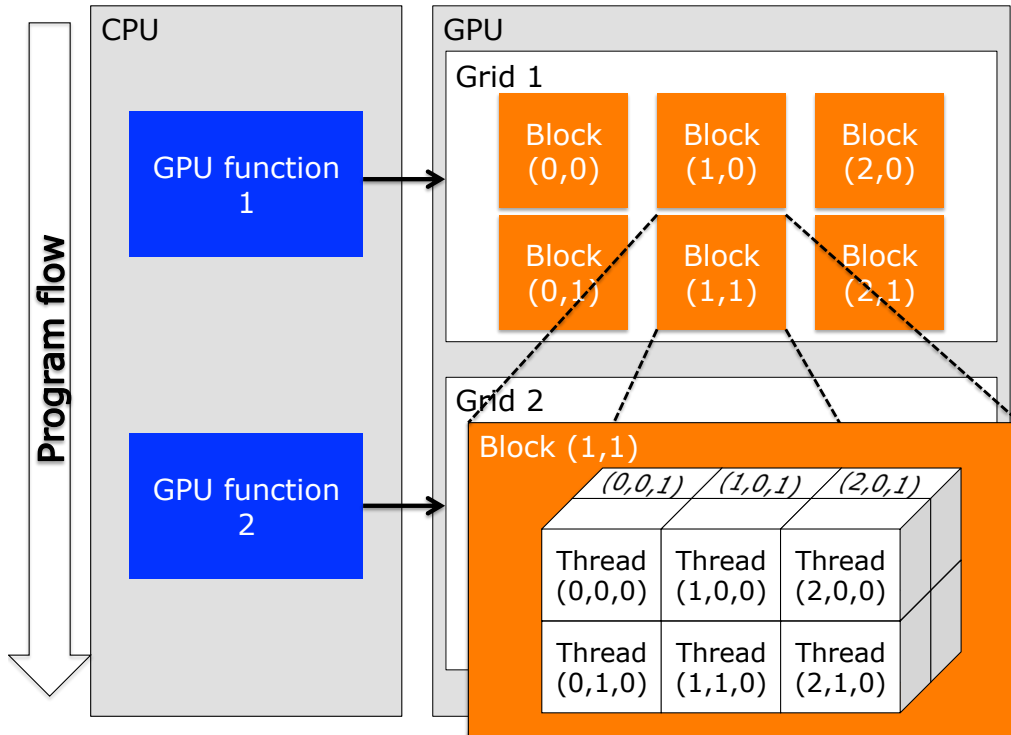


Figure 1.1: Mapping of threads, blocks, and grids in CUDA.

## CUDA Streams

Multiple CUDA operations can be performed simultaneously using CUDA streams [6]. A stream is defined as a sequence of operations executed in an issued order on the GPU, allowing multiple memory copies to be operated between the CPU and GPU or multiple GPU functions to be launched simultaneously. Although CUDA operations issued for a specified CUDA stream, such as memory copies and execution of GPU functions, are guaranteed to occur in the order of the issued operations, the order in which the operations issued to other CUDA streams are executed is not restricted; in fact, they are executed simultaneously to the maximum extent allowed by the GPU resources.

Table 1.2: Execution times for resizing an image using texture memory (source image resolution:  $640 \times 480$ )

Target resolution	$160 \times 120$	$320 \times 240$	$1280 \times 960$	$2560 \times 1920$
Hardware	0.0603 ms	0.0697 ms	0.3337 ms	0.7244 ms
Software	0.0634 ms	0.0784 ms	0.4681 ms	1.2793 ms

### Texture and Shared Memories

The CUDA provides different types of memory and allows programmers to use preferred memory areas. Global memory is the standard memory that can be read and written by a CUDA thread. Shared memory can be used as a scratch pad memory to temporarily share data among threads. In addition, texture memory exhibits the following features:

- read-only access from CUDA threads,
- fast access from CUDA threads, and
- interpolation by hardware—a remnant from the original design of GPUs for graphics processing.

Table 1.2 shows the computing times required to resize the video graphics array (VGA)-sized ( $640 \times 480$ ) images by bilinear interpolation. In the table, *Hardware* indicates the case where the interpolated values are automatically computed on the hardware using texture memory, whereas *Software* represents the cases where the interpolation program is implemented manually and computed as a GPU function. Regardless of the image size, the computational speed is higher when hardware interpolation based on texture memory is employed. The texture memory can be set to return an interpolated value automatically when accessed by specifying the pixel coordinates in an image [7]. This ability reduces the load on the SPs and enables an efficient conversion because the values are automatically interpolated by the GPU.

## 1.2. Contributions and structure of this study

---

Table 1.3: General characteristics for various types of external sensors leveraged in autonomous driving

sensor	typical data source	advantages & disadvantages	typical usage
camera	visible ray	<ul style="list-style-type: none"><li>• reasonable cost</li><li>• spatially dense data</li><li>• affected by weather conditions</li></ul>	detection / tracking / recognition
LiDAR	laser	<ul style="list-style-type: none"><li>• precise 3D data</li><li>• long sensing range</li><li>• high cost</li><li>• spatially sparse data</li></ul>	detection / tracking / localization
radar	radio wave	<ul style="list-style-type: none"><li>• reasonable cost</li><li>• working well at night and bad weather</li><li>• affected by target reflectance</li></ul>	detection
GNSS	satellites	<ul style="list-style-type: none"><li>• directly acquires location</li><li>• unavailable at indoor and tunnels</li></ul>	localization

---

## 1.2 Contributions and structure of this study

To achieve high-level autonomous driving systems, this study focuses on one of the three primary elements of autonomous driving: perception tasks. To determine ego vehicle behavior, including acceleration, deceleration, turn, and lane change, high-level autonomous driving systems must perceive the surroundings and circumstances of all directions using external sensors.

Because each type of sensor has its own strengths and weaknesses, various types of sensors are typically leveraged to increase the overall strength. Representative sensors include cameras, LiDARs, radars, and GNSSs, and their general characteristics are listed in Table 1.3. The following perception tasks are typical examples where these sensors are used:

**Detection:** Vehicles and/or humans around the ego vehicle are detected to avoid collision with them. Typically, data captured by cameras or LiDARs are used to perform this task.

**Tracking/Prediction:** Tracking refers to assigning the correspondence between objects detected in the current frame and those detected in the previous frame for predicting their movement in the future frame. The prediction results for the movement of other objects is used to plan the ego vehicle behavior. In addition to the prediction of object movements, this task also includes the prediction of object locations. Location prediction is occasionally introduced into perception pipelines to interpolate the temporal continuity of object locations when detection tasks require a significant amount of time to complete.

**Recognition:** This task provides understanding regarding traffic information provided to human drivers, such as traffic lights and traffic signs (i.e., classifying the type of information to be adhered by the ego vehicle). Visual information, such as images captured by cameras, is often used for this task.

**Localization:** This task identifies the location of the ego vehicle. To control the ego vehicle precisely, this task is essential.

Hence, the perception tasks for autonomous driving are a group of cooperative tasks rather than a single type of task for understanding the driving environment and determining the ego vehicle behavior.

Object detection is a typical perception task where objects such as vehicles or pedestrians are identified from raw or preprocessed sensor data. Although some recent studies [8–11] have performed object detection using non-image sources, image-based object detection remains essential owing

to the wide adoption of camera sensors, reasonable cost, and ability to acquire spatially dense data. The deformable part model (DPM) [12] is a representative object detection algorithm that was used to achieve high detection accuracies in early studies. Because advanced driver-assistance systems, which are widely implemented in many commercial vehicles, are typically based on traditional pattern recognition techniques, the approaches for these traditional techniques are expected to provide insights into the development of autonomous driving systems. Despite its high detection performance, its computational cost prevents its practical usage; this tradeoff between detection accuracy and computational cost (i.e., execution time) is one of the primary problems to be addressed for object detection tasks.

Recognizing traffic light states is another perception task that arises as a characteristic task of autonomous driving. Autonomous driving vehicles are expected to share roads with vehicles driven by people during the transition period for their introduction, and autonomous vehicles must be able to recognize a wide range of traffic information, such as traffic lights. Because traffic lights installed on public roads are designed for the visual perception of humans, image-based methods are the first option to manage this task. However, images captured via vehicle-installed cameras typically contain various objects unassociated with traffic light state recognition, which often cause misrecognition. For autonomous driving systems, map information is frequently used as prior knowledge of the driving environment. Particular maps used in autonomous driving systems are referred to as *high definition maps*, which can be categorized into two types: *point-cloud maps*, which comprise a set of points (typically stored in three-dimensional coordinates) representing the surface in the driving environment; and *feature maps*, which are a series of information for road features, including poses (i.e., three-dimensional (3D) position and orientation) of traffic lights, traffic signs, and utility poles, as well as information pertaining to lanes where autonomous driving vehicles can be driven. By intuition, leveraging these high definition maps will improve image-based traffic light recognition because the systems can perform recognition based on their location on the maps, the direction they are facing, and the potential locations of traffic lights.

In addition to perception algorithms, computation offloading and system

scaling should be addressed for the actual deployment of autonomous driving systems. To ensure automated driving safety, multiple external sensors are typically installed in such systems. For example, if a system comprises six full-HD cameras and five LiDARs, the amount of data for one frame (i.e., data amount fed at a time) will be  $1920 \times 1080$  (pixels)  $\times 3$  (channels)  $\times 6 \approx 35$  (MB) from cameras and approximately  $200,000$  (points)  $\times 4$  (channels)  $\times 5 \approx 4$  (MB) from LiDARs when considering the amount of raw data for brevity. Furthermore, the amount of data increases linearly with the number of external sensors, which is a straightforward strategy to eliminate blind spots. If these large volumes of data concentrate on a single unit, then the computational and/or data transfer resources might saturate by only processing perception tasks; this prevents the system from operating as intended. Meanwhile, if decentralized processing is introduced to the system to distribute the computational load, then additional latency caused by its introduction should be considered; however, the effect is unclear and developers are hesitant to adopt this approach.

For tasks involving data parallelism, GPUs are considered a promising method for accelerating computations. Because of the continuous advancements in programming languages such as the CUDA, GPUs are rapidly evolving into general-purpose computational devices. However, because the performance characteristics of GPUs are diverse [13], the validity of using GPUs for specified tasks depends on the type of task, and it is necessary to assess whether these tasks can benefit from GPU usage. Moreover, traditional GPUs are known to be massively parallel computing devices that consume significant quantities of power. In particular, power consumption should be considered when deploying autonomous driving because a power supply equipped for on-board systems is typically limited and not intended to cover large power consumption.

The main contribution of this study is exploring methods to accelerate perception tasks for autonomous driving using GPUs to fulfill the criteria for high-level autonomous driving systems. Because the application of GPUs to perception tasks for autonomous driving requires the consideration of several issues, including validity to the tasks, power consumption, and effective utilization, the following three research topics were investigated in this study:



Topic 1: Object detection acceleration using GPUs, including the confirmation of GPU validity for traditional pattern recognition tasks.

Topic 2: Practical verification of a scheme for traffic-light-state recognition for autonomous driving using GPUs, and

Topic 3: Performance analysis of a prototype system for decentralized processing using embedded-oriented GPUs.

In this study, the three topics mentioned above were investigated to solve/reveal the aforementioned problems regarding autonomous driving perception. Although, it is noteworthy that other topics, including object detection from 3D data or the acceleration of planning algorithms, are equally valuable for achieving high-level autonomous driving systems, topics other than the above three are out-of-scope of this work.

## 1.3 Organization

The remainder of this dissertation is organized as follows: Relevant studies are summarized in Chapter 2. The validity of using GPUs for specific autonomous driving tasks is discussed in Chapter 3 and 4. In Chapter 5, problems that arise when applying GPUs to on-board autonomous driving systems and a possible solution are discussed; additionally, an analysis of a prototype system for performing one of the perception tasks using GPUs and the practical usage of GPUs in the actual deployment of autonomous driving are presented. Some topics associated with this study are presented in Chapter 6, and the conclusions are summarized in Chapter 7.

# Chapter 2

## Related studies

### 2.1 Object detection

#### 2.1.1 Traditional algorithms

Object detection is a fundamental challenge in intelligent applications, including autonomous driving, and has been a primary area of research in the field of computer vision for many years. Prior to 2000, mainstream object detection techniques involved manual designs and utilization of image content representations as detection objectives. Scale-invariant feature transform (SIFT) [14], Haar-like [15], and the histogram of oriented gradients (HOG) [16] are some of the popular representations used. The deformable part model (DPM) [12], which is based on the HOG, is a typical object detection algorithm that was used to achieve high detection accuracies in early studies. As spreading recognition of its performance, the computational cost of DPM has been highlighted as a significant limitation to its practical use. The cascaded DPM [17] was introduced to reduce the computation time of the original DPM via a cascade detector, which omits computations for areas that would not include target objects; this approach improved the speed of the DPM by 20 times compared with the model presented in [18]. For example, the mean detection time per frame for a person model reduced from 8.5 s to 682 ms. Yan *et al.* extended the cascaded DPM to account for the relationships between neighboring detection windows,

while introducing look-up tables of the HOG features to further reduce the computational time [19]. Pedersoli *et al.* also reduced the computation time of the DPM by introducing a hierarchical model of image resolutions to reduce the area of an image for the detector [20], thereby increasing the detection speed by approximately two times compared with that achieved by [17], with a marginal decrease in the detection accuracy. Cho *et al.* focused on automotive applications to which DPM is applicable and introduced geometric constraints to improve the detector of the DPM, thereby achieving higher accuracy and faster computation [21]; specifically, a detection rate of 14 fps for a  $640 \times 480$  sized image, although only on pedestrian detection under specific conditions was emphasized. Dean *et al.* applied a hash technique to the look-up tables used in the DPM and enabled up to 100,000 object classes to be detected on a single machine [22]; they claimed that the speed improvement achieved in the DPM was almost comparable to that reported in [17].

The abovementioned studies contributed to the algorithmic improvement of the DPM. However, the focus of this study is the original DPM algorithm, with the exploration of the use of massively parallel computing with GPUs and characterization of the DPM computations. Because the acceleration of an algorithm using a GPU does not change the algorithm theoretically, one of the advantages of the proposed method is the absence of accuracy loss caused by algorithmic modification. More details regarding such acceleration and its evaluation results are presented in a later chapter.

### 2.1.2 Cutting-edge methods using revived neural network techniques

Owing to the successful application of deep CNNs in a competition for recognizing large-scale general scenes [4, 23], the mainstream of object detection has shifted toward the design and utilization of deep CNNs. GPUs that provide massively parallel computing and various large-scale datasets, including PASCAL VOC [24], MS COCO [25], ImageNet [23], and KITTI [26], have been instrumental in this shift. Compared with humans, CNNs can generate more complicated representations, and these

representations (also referred to as *features*) have high expressiveness, thereby affording high accuracies.

You only look once (YOLO) [27] and the single-shot multibox detector (SSD) [28] are some of the popular detectors based on the deep CNN. Their advantages include the combination of classifications and bounding-box regressions into a single network to accelerate the inference time. These methods and succeeding studies have inspired the realization of accurate object detection in real time. As out-of-scope of this study but related ones, object detection using point clouds from LiDARs has been reported recently [8–10]. In these methods, object positions and orientations are directly output in the coordinate system in which the planning of the ego vehicle is performed. Because point clouds are unstructured data compared with images, neural networks in these methods are leveraged to extract features, followed by the conversion of the extracted features into forms manageable by two-dimensional CNN-based object detectors, such as the bird’s eye view. In another study [11], *graph neural network* techniques were leveraged on point clouds to manage unstructured data directly, although the inference speed can be further improved despite its good accuracy.

However, the complicated representations generated by neural networks may complicate the improvement in accuracies and the analysis of error causes. Owing to the increased utilization of deep CNNs, researchers have focused more on improving their interpretability. For example, gradient-weighted class activation mapping (Grad-CAM) [29] approach helps one to visualize the basis of the network’s responses via heat maps that show the contributions of different regions to responses. In other studies [30, 31], the response reasons of deep neural networks to autonomous driving were visualized. Although the interpretability of the deep CNN is beyond the scope of this study, it is one of the key considerations for the further development of autonomous driving systems.

## 2.2 GPUs as decentralized units

NVIDIA Corp. has introduced the Jetson series of GPUs, which satisfy various requirements, including computing performance and power consumption

## 2.2. GPUs as decentralized units

---

Table 2.1: Comparison between several products from the Jetson series<sup>†</sup>.

	TX1	TX2	AGX Xavier	Nano
Release year	2016	2017	2018	2019
# of GPU cores	256	256	512	128
# of Tensor cores	—	—	64	—
GPU architecture	Maxwell	Pascal	Volta	Maxwell
Memory amount	4 GB	8 GB	16 GB	4 GB
Power	Under 10 W	7.5 W/ 15 W	10 W/15 W/ 30 W	5 W/10 W
DL accelerator	—	—	2× NVDLA engines	—

<sup>†</sup> Partially cited from <https://developer.nvidia.com/embedded/develop/hardware> and modified.

limitations. Table 2.1 shows a comparison of the specifications of several products from the Jetson series. This series of products has garnered attention for its high computing performance and low power consumption, and some relevant studies have been published.

The authors of [32] investigated the porting of deep learning (DL)-approaches to a small and power-efficient device, focused on pedestrian detection using DL, and analyzed the suitability of the Jetson TX1 mounted on a mobile robot in comparison with a high-performance GPU (GTX Titan X). Furthermore, the authors also investigated the effects of changing the operation precision used in the CNN for pedestrian detection from the typical 32-bit floating point to 16-bit floating point type, which resulted in a 15× processing acceleration. Meanwhile, the authors of [33] proposed an unmanned aerial vehicle (UAV) warning system using on-board and real-time object detection. To fulfill the requirements of the system, such as minimal power consumption, limited on-board processing power, and minimal weight, the Jetson TX2 was employed. In another study, the authors of [34] evaluated

various algorithms on *visual simultaneous localization and mapping* (visual SLAM) to estimate the self-localization capabilities of mobile robots using the Jetson TX2; they reported that the CPU load can be decreased and the processing speed can be increased by employing an embedded GPU.

In addition to these studies, several studies regarding on-board processing systems for perception tasks using the Jetsons products have been reported. These studies primarily focused on the processing time of a single task executed on a Jetson, i.e., embedded-oriented GPUs were used as the host processors of the systems; however, such units are insufficient as hosts for managing fully autonomous driving systems. Nevertheless, embedded-oriented GPUs can be considered for application in autonomous driving systems as edge devices, whereas their effects on system integration, including delays caused by their introduction as decentralized processing platforms, remain unclear. In a later chapter, the effect of introducing the Jetson AGX Xavier as a decentralized processing platform is explored, as well as the validity of decentralized processing on the perception of autonomous driving systems.

## Chapter 3

# GPU-Accelerated Object detection

### 3.1 Accelerating traditional object detection algorithm using GPUs

Object detection is a fundamental challenge in intelligent applications, including autonomous driving. Since moving objects have non-uniform shapes, complex algorithms are often required to recognize and classify them. Although various sensors are available for object detection, one of the most widely deployed and reasonable devices is a camera. Commercially available cameras provide high-definition images containing meaningful information. Recent trends in object detection have exploited *sensor fusion* techniques, where data from different types of sensors are systematically fused to obtain more information. With these techniques, however, cameras are mostly needed for object detection.

Before the popularity of CNNs, the DPM algorithm [12] based on the HOG [16] was dominantly used for image-based object detection. The HOG is known to provide strong feature descriptions representing the gradient orientations extracted from locations on a grid superposed on an image. Complex moving objects, including cars and pedestrians, can be adequately featured with the HOG even under bad weather conditions. The DPM had demonstrated higher detection rates than existing algorithms of those

days in the Visual Object Class Challenge [24], which is an international competition on object recognition technologies and logic hosted by PASCAL, the image processing community in Europe. Although DPM demonstrated high detection rates under various conditions [24], it is constrained by high computational cost.

Advanced driver-assistance systems (ADASs) are widely implemented in many commercial vehicles these days. Although these systems are categorized into or under level two among the options in Table 1.1, the technologies developed with such systems are expected to provide insights into developing higher-level autonomous driving systems. ADASs that support driver perceptions, including object detection, are typically based on traditional pattern recognition.

In this section, a precise workflow analysis and GPU implementations of a traditional pattern recognition algorithm for object detection are first presented; then, an example of the effects of applying GPUs to perception tasks is discussed as a preliminary study.

### **3.1.1 Preliminary analysis of DPM workflow**

#### **Deformable Part Models**

The main characteristics of a DPM are summarized as follows: 1) flexible recognition of vehicles with different wheel locations and body lengths, 2) vehicle recognition from any direction, and 3) recognition of various objects with high success rates.

These characteristics are realized by flexibly incorporating similar local features and their positional variations as part of the overall information describing an object. Specifically, DPMs use the HOG [16] features with two types of filters (root and part filters). The HOG features are robust to illumination and local shape changes since color information is not required, and the feature quantity in each local area is normalized. The root filter stores the features of the entire object as a model, whereas the part filter stores the features of a part of an object. Furthermore, the input images are resized to create a pyramid structure to accommodate the sizes of the recognized objects in an image. The DPM is schematically depicted in



### 3.1. Accelerating traditional object detection algorithm using GPUs

---

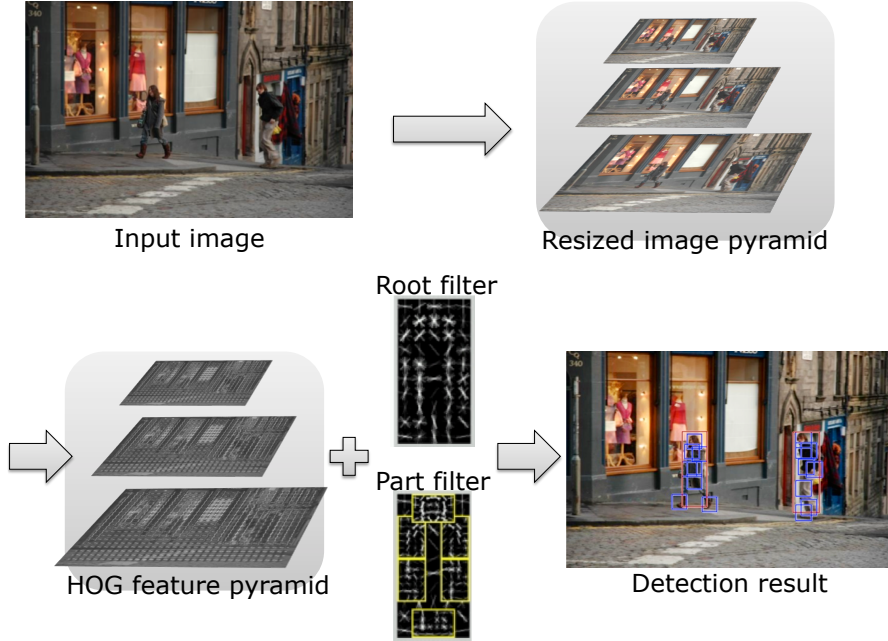


Figure 3.1: Overview of the DPM algorithm [18, 35].

Fig. 3.1.

Although the DPM theoretically provides high detection rates, its execution time caused by its high computational cost is often problematic. Several studies [17, 20–22] have demonstrated improvements to the speed of the DPM. However, the execution time of the DPM still remains an open problem for runtime usage; in fact, an open-source code for fast DPM implementation is not available. As a result, the real bottlenecks to accelerating DPM computations and approaches to eliminate them are unknown.

#### Workflow Breakdown

Fig. 3.2 shows the details of the sequential execution time for a VGA-sized ( $640 \times 480$ ) image when the original DPM algorithm, implemented in C++ with a car model specified in PASCAL VOC 2007, was executed on an Intel Xeon CPU E5-2687W v2. The execution times are averaged over ten samples.

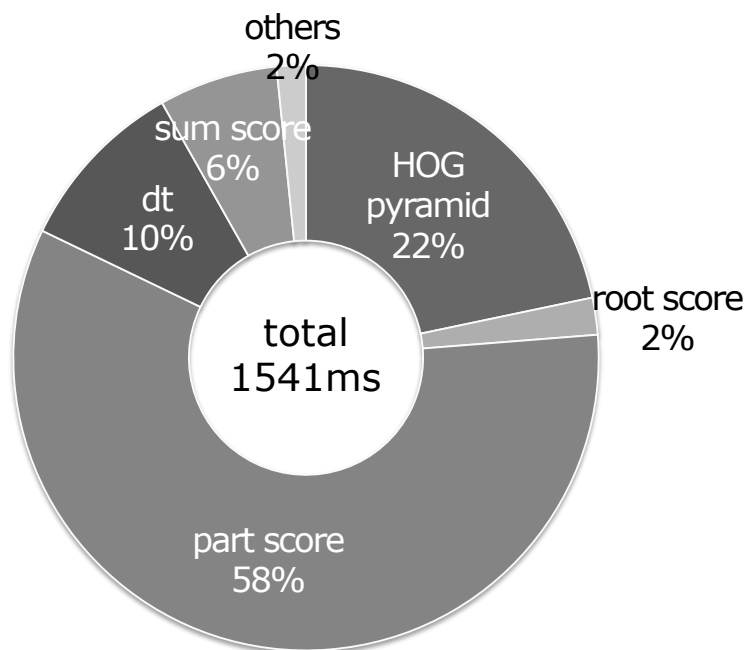


Figure 3.2: Breakdown of the execution time of a DPM implemented on a single-threaded processor.

This DPM implementation requires more than 1500 ms per frame. Based on the DPM analysis, it was found that approximately 98 % of the execution time was spent on the following five processes:

#### HOG pyramid:

The input images are resized, and a quantity of the HOG features is computed for each image to generate the “image feature pyramid”.

#### Root score:

Convolutions are computed between the root filter and detection windows, which are partial images of the HOG features; the output values of these computations are called “root scores”, which represent the similarities between the HOG features and root filter.

#### Part score:

This step involves the same computations as those for the root filter,

### 3.1. Accelerating traditional object detection algorithm using GPUs

---

with the exception that the part filter is applied; the output values of these computations are called “part scores”, which represent the similarities between the HOG features and part filters.

dt:

Distance transforms [36] are obtained to sum up the scores from the part filters. Since the DPM considers positional variations of the parts of an object feature through the part filters, the best positions for the part filters are identified at this stage.

Sum score:

The scores from the root and part filters are combined by means of the dt to obtain a final score, which is compared to a threshold to determine whether the given object is the recognition target.

In the program code, these processes are expressed by a loop structure with iterative computations. The computation results from each iteration are mostly independent of the results of other iterations. This indicates that the program code of the DPM is data parallel and that GPUs are suited to improve the computational speed for DPM. However, an appropriate method to implement DPM on a GPU has not been suggested. Therefore, possible schemes for implementing DPM on the GPU are explored, and the impact of GPUs on DPM computations are evaluated.

#### 3.1.2 GPU-based implementation of DPM

This section presents some schemes for GPU-based implementation of the DPM. First, multiple parallelization schemes are proposed as guidelines for implementing DPM on the GPU. In these schemes, the logic of the DPM algorithm remains the same as that of the original implementation [18]. For more details on the DPM, such as the equations used, readers are encouraged to refer to the original work [18].

The proposed parallelization schemes are based on *loop unrolling*. Fig. 3.3 illustrates an example of unrolling for nested loops. To apply this loop unrolling, data parallelism is basically required for the expressions executed in the loop; the variables in the expressions executed iteratively have to

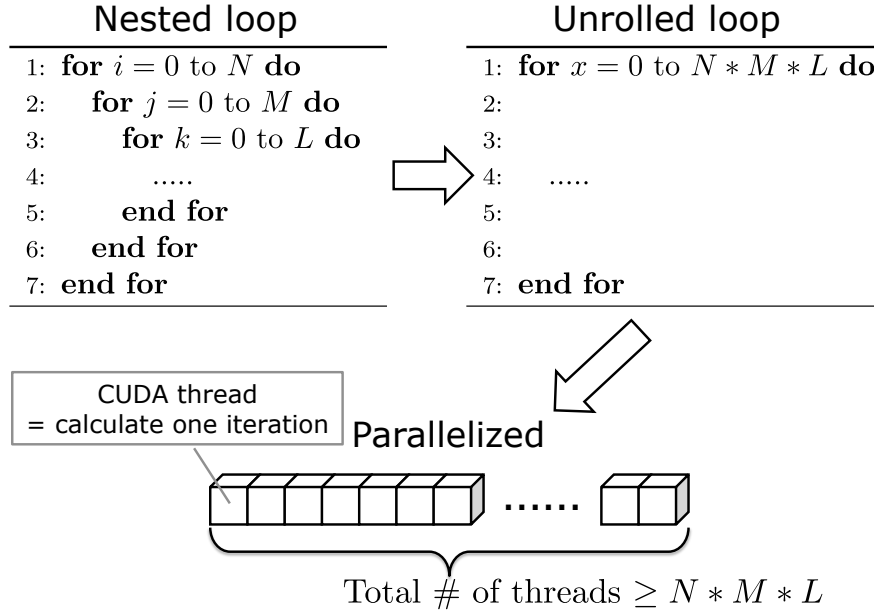


Figure 3.3: Concept of loop unrolling.

be independent over iteration (i.e., the variables at any iteration times are not affected by other iterations). Since the aforementioned five processes of DPM were revealed to have data parallelism by detailed workflow analysis, it is possible to apply this loop unrolling and expected to be accelerated their computation. Multiple factors are involved in loop unrolling; if the loops are unrolled in the same manner as shown in Fig. 3.3, this algorithm can be parallelized on a GPU using  $N * M * L$  threads. Another possible scheme is that all the loops are *not* unrolled; instead, only the innermost loop ( $k = 0$  to  $L$ ) to be offloaded to the GPU is unrolled, while the outer loops are iterated on the CPU. A hybrid between these two schemes can also be used to balance the tradeoffs between CPU and GPU workloads. These strategies are available because the expressions in the loop are independent over iterations. Therefore, loop-unrolling schemes can be classified into one of the following three categories:

- Unrolling as many nested loops as possible and possibly producing idle

### 3.1. Accelerating traditional object detection algorithm using GPUs

---

threads on the GPU, depending on the number of iterations in the nested loops.

- Unrolling only the innermost loop, and possibly causing the CPU to execute many threads that would compete for GPU resources.
- Unrolling multiple inner loops, namely a hybrid between the above two schemes, and relaxing the number of threads on the CPU while avoiding many idle threads on the GPU.

The remainder of this section is devoted to the proposed parallelization schemes based on loop unrolling. All these schemes utilize the texture memory on the GPU. The shared memory on the GPU is not encouraged for the proposed implementation as the size of the input data is much larger than that of the shared memory, and the read-only data can be stored in the texture memory. Single-precision floating-point values are used for input data.

#### Root and Part Scores

Under the CUDA programming model, a block with a lot of idle threads can degrade the performance of a GPU. The key to maximizing the performance, therefore, is to retain the maximal number of active threads in each block.

For example, computation of the root and part scores, which consume the most time in a DPM, as demonstrated in Section 3.1.1, is schematically depicted in Fig. 3.4. Note that the root and part scores are obtained using the same algorithm, although the areas to which the target objects are applied are different. A total of  $M$  filters is applied to  $N$  resized HOG feature images, producing  $N \times M$  score arrays. Note that the widths and heights of the score arrays differ in reality with the  $z$  value ( $1 \leq z \leq N \times M$ ), and the number of elements on the  $x$ - $y$  plane, where computations are performed, depend on the value of  $z$ .

Data parallelism of each element enables the generated score arrays to be efficiently computed on the GPU using CUDA threads. However, if  $N \times M$  is mapped in the  $z$  direction of the grid, the score arrays must be mapped toward the maximum width (`max_width`) and maximum

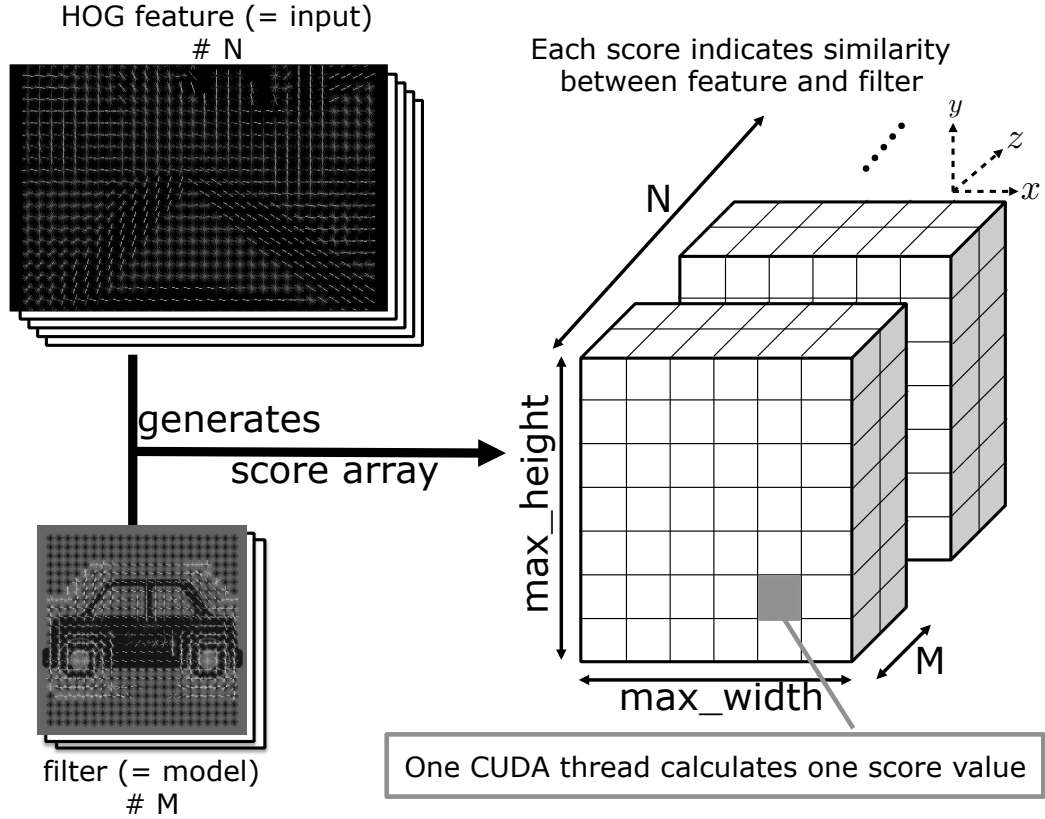


Figure 3.4: Computation of a score array in DPM.

height (`max_height`) in the  $x$  and  $y$  directions of the grid, respectively, to compute the scores. This direct mapping of the scores to the CUDA threads corresponds to a DPM implementation that loosely unrolls all the computational loops of the iterative scores, which is termed *loosely unrolled* parallelization hereafter. As shown in Fig. 3.5, the disadvantage of the loosely unrolled parallelization scheme is that it may produce a large number of idle threads within a block owing to variations in the widths and heights of the  $x$ - $y$  planes, thereby preventing the maximum utilization of the GPU.

To prevent an excessive number of CUDA threads, the CPU threads are utilized with CUDA streams, although the aforementioned DPM implementation is based on native mapping of the CUDA threads and score arrays. Specifically, in the proposed DPM implementation, a dedicated CUDA stream is assigned to each CPU thread on which the loosely unrolled parallelization is applied individually. This hybrid scheme using both GPU

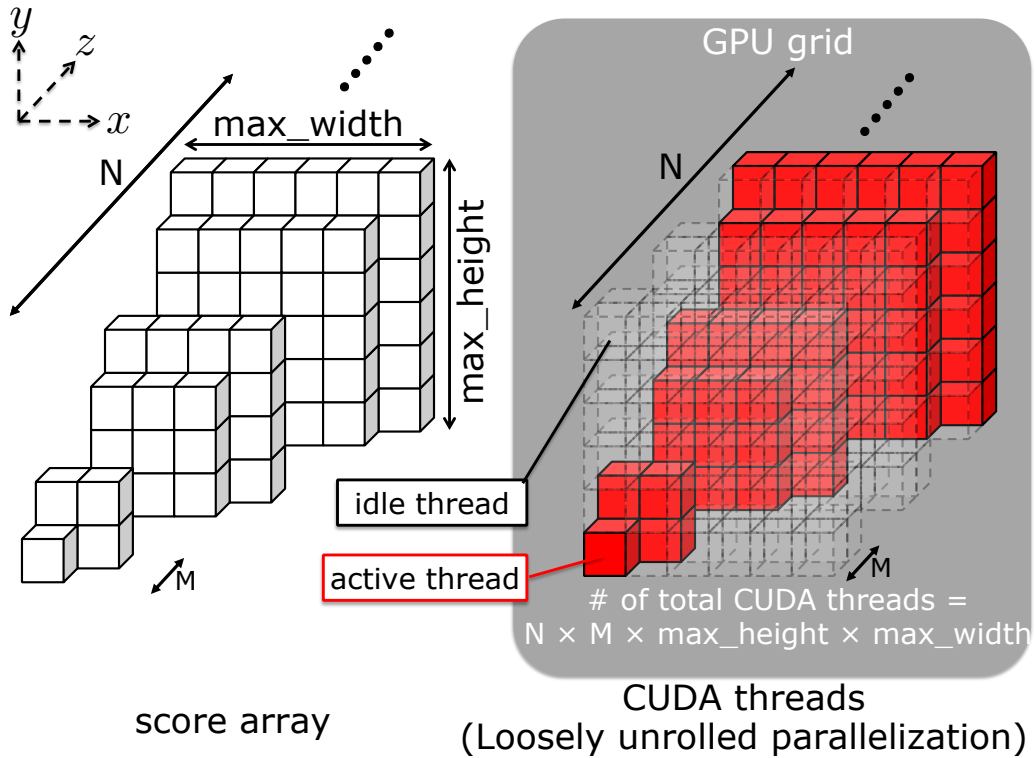


Figure 3.5: Loosely unrolled parallelization of a score array.

and CPU threads is hereafter called *tightly unrolled* parallelization.

A schematic of the score array under tightly unrolled parallelization is shown in Fig. 3.6; this scheme allows only cubes of the same size to be unrolled together so that the idle threads within a block can be removed. Although the utilization of GPU resources is optimized, the downside to tightly unrolled parallelization is that it imposes a large number of CPU threads on the host processor. For example, the number of CPU threads required in the case shown in Fig. 3.6 is  $N \times M$ . Specifically, in this study, the maximum values of  $N$  and  $M$  are 25 and 12, respectively, such that the total number of CPU threads is 300. These values derived from the runtime configuration, i.e., how many HOG images were calculated and how many filters were applied. Since the CPU is not designed to execute a large number of threads simultaneously, the speed of the resulting DPM computations may degrade owing to overloading of the CPU capacity.

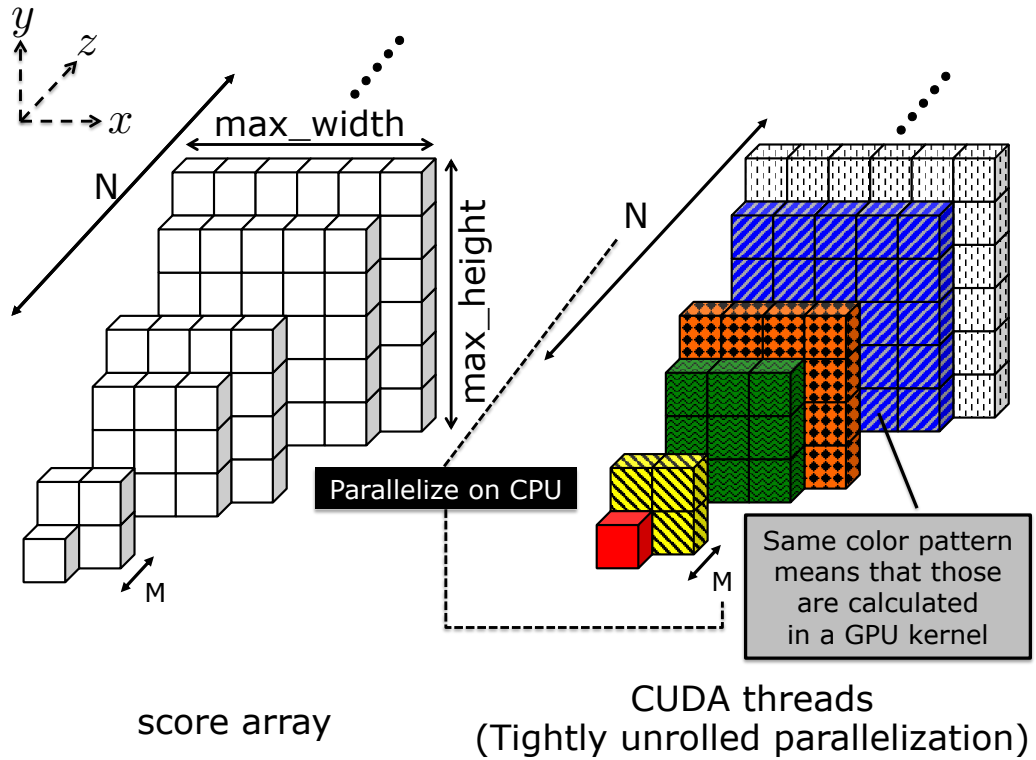


Figure 3.6: Tightly unrolled parallelization of a score array.

The tightly unrolled parallelization scheme is improved in a heuristic way, so full use of the GPU resources is compromised, but a balancing point can be found between the CPU and GPU threads. This new parallelization scheme called *hybrid parallelization* applies loosely unrolled parallelization partly to a unit of filters ( $M$ ), as shown in Fig. 3.7. This hybrid scheme increases the number of idle threads on the GPU while reducing the number of CPU threads to  $N$  at most, thereby relaxing the limitations of tightly unrolled parallelization.

### HOG Pyramid

Constructing the HOG pyramid is another bottleneck in the computation of the DPM and comprises four stages: (i) resizing the input image, (ii) generating histograms of the resized images, (iii) generating the norms, and (iv) calculating the feature vectors of the images. Each stage of (ii) ,



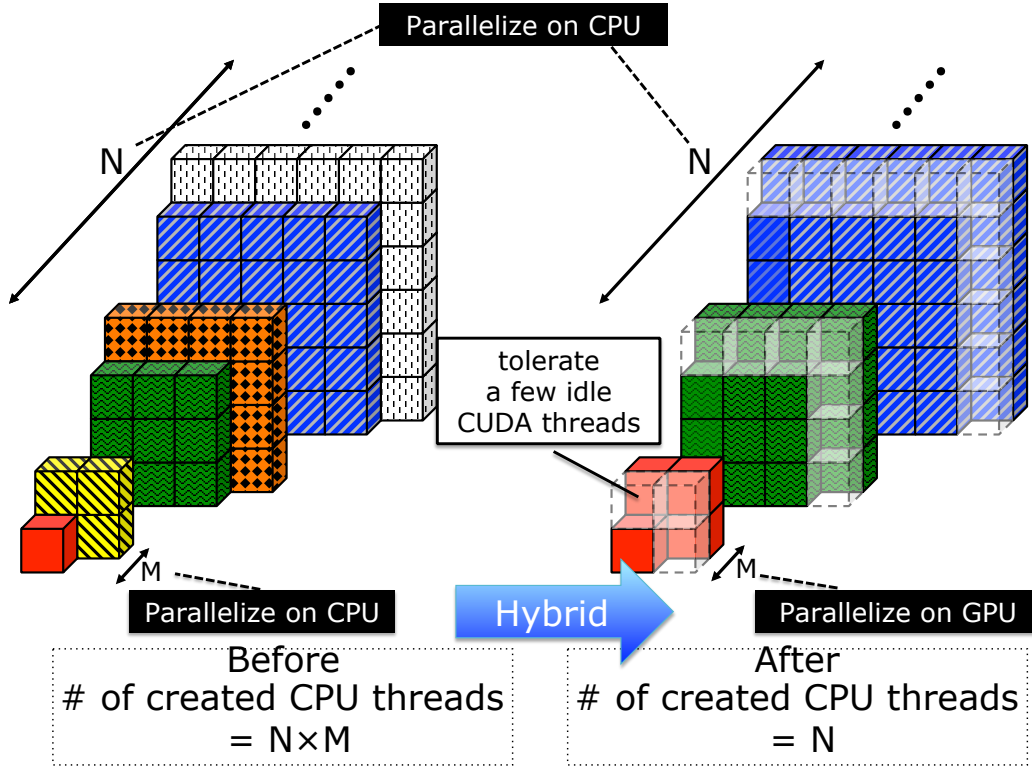


Figure 3.7: Hybrid parallelization of the score array.

(iii), and (iv) can be applied to each resized image and parallelized using a large number of threads. Hence, each stage can be implemented as a GPU kernel, and CPU threads can be created for each resized image; that is, the CUDA streams are assigned individually to launch all the GPU kernels simultaneously. A parallelization scheme for constructing the HOG pyramid is shown in Fig. 3.8.

To reduce the computation time when resizing the input image, the original DPM implementation [35] replaces the actual numerical computations, which obtain pixels of the resized images, with references to the static interpolated values. This approach is reasonable for sequential processing; however, a bilinear algorithm is used for the proposed GPU-based DPM implementation because it provides the following features suited for GPU programming: highly data-parallel computation, simple computational logic, and hardware interpolation capability.

In particular, the third feature of the bilinear algorithm allows the use of

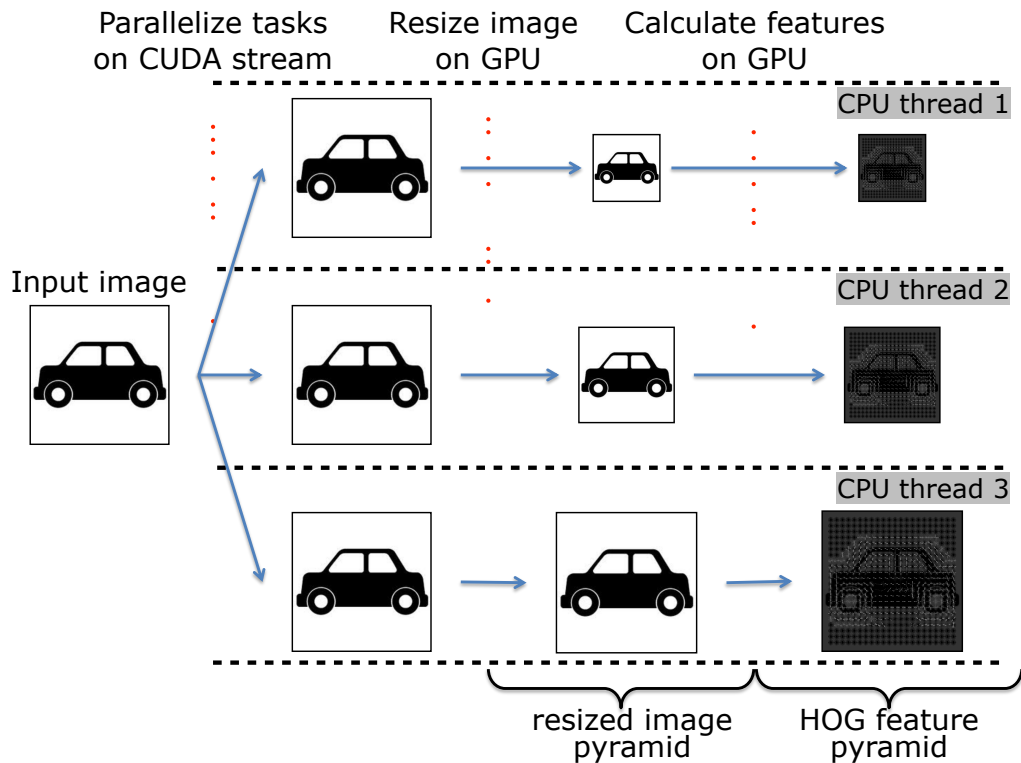


Figure 3.8: Approach for constructing a HOG pyramid.

a hardware interpolation function of the GPU texture memory, as described in Section 1.1.2.

Fig. 3.9 depicts the three implementation schemes for constructing the HOG pyramid. Two variants of the GPU implementation are provided using loosely and tightly unrolled parallelization schemes. Unlike the root and part scores, all computational blocks for each of the resized images have a common  $x$ - $y$  plane, so hybrid parallelization need not be considered.

### Distance Transforms and Score Summation

Listing 3.1: Pseudo-code structure of the distance transforms in DPM

```

1  for (level = intervals; level < L_MAX; level++) {
2      ...
3      for (jj = 0; jj < Num_of_components; jj++) {
4          ...
5          for (kk = 0; kk < numpart[jj]; kk++) {
    
```

### 3.1. Accelerating traditional object detection algorithm using GPUs

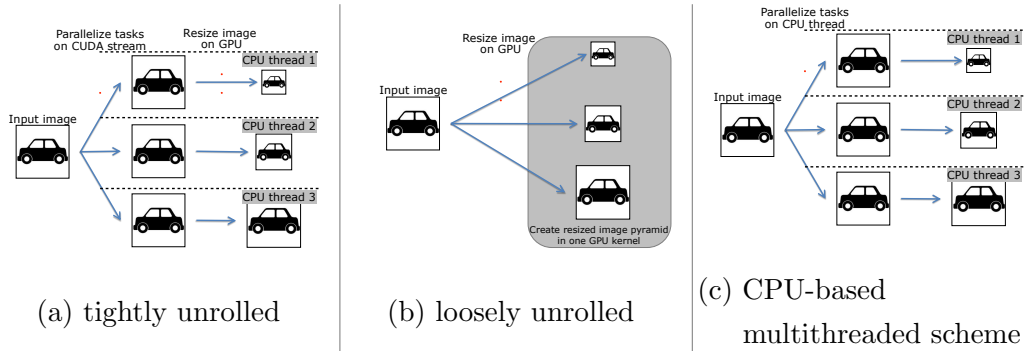


Figure 3.9: GPU-based implementation schemes using the bilinear algorithm, with tightly unrolled (left) and loosely unrolled (center) parallelizations compared to a CPU-based multithreaded scheme (right).

```

6      /*****/
7      /* Distance transformation */
8      for (x = 0; x < dims[1]; x++) {
9          ...
10     }
11     for (y = 0; y < dims[0]; y++) {
12         ...
13     }
14     /* Distance transformation */
15     /*****/
16 } // for (kk)
17 } // for (jj)
18 } // for (level)

```

Listing 3.1 is an overview of the pseudo-code for the distance transforms (`dt`), which are performed by the 8th to 13th lines of code represented by two consecutive loops. After computing the responses of the part filters to the HOG features, spatial uncertainty is considered to the part filter responses by this transformation. Briefly, this transformation searches the highest response of the filters from neighbor locations considering deformation cost (i.e., how far the highest response locates from the current position) [18]. More details are described in [18, 36, 37]. The three outer loops repeat the distance transforms; hence, the `dt` section of the algorithm contains

quadruple loops. Computations in each iteration of these loops exhibit data parallelism; therefore, all the loops in Listing 3.1 are unrolled.

Listing 3.2: Pseudo-code structure of the score summations in DPM (simplified)

```
1 // Get where calculation should be started
2 const int y_base = ...;
3 const int x_base = ...;
4 for (ii = 0; ii < Image_width; ii++) {
5     ...
6
7     for (jj = 0; jj < Image_height; jj++) {
8         ...
9
10        float score = score[...]; // get compared score
11        int y = y_base + jj;
12        int x = x_base + ii;
13        ...
14
15        float *target = ...; // get update location
16        if (score > *target) {
17            // update value according to the condition
18            *target = score;
19        }
20    } // for (jj)
21 } // for (ii)
```

Listing 3.2 shows the structural overview of the pseudo-code for the score summations (`sum score`). Since the structure of the score summation code is similar to that for the distance transforms, the same scheme (that is, unrolling all the loops and parallelizing all the iterations on CUDA threads) is adopted.

### Leveraging Texture Memory

As described in Section 1.1.2, the texture memory is constrained by read-only access, but the access speed is significantly greater than that of the global memory. Therefore, the texture memory is used only for data that are referenced by CUDA threads (i.e., never modified). Table 3.1 shows the

### 3.1. Accelerating traditional object detection algorithm using GPUs

---

Table 3.1: Data that can be stored in the texture memory.

DPM Part	Target Data
HOG pyramid (create resize image pyramid)	input image
HOG pyramid (create HOG pyramid)	resized image pyramid
root/part scores	HOG features, model filter

data used in the DPM that can be stored in the texture memory. The proposed GPU-based DPM implementation also uses texture memory for these data. To leverage the texture memory, mainly two steps are required; replacing the copy destination from the global memory on GPU to the texture memory when moving data from CPU to GPU, and switching memory access operations to the dedicated fetch operations. By these simple modifications, CUDA threads enjoy the benefit of the fast data fetching from the texture memory on GPU. If data type stored on the texture memory is supported one (e.g., `float` in CUDA C++), automatic (hardware implemented) data interpolation is also available. As an example of the automatic data interpolation using the texture memory, Listing 3.3 shows code snippets in CUDA C++ to perform bilinear resizing of the input image, which corresponds to the first row (“HOG pyramid (create resize image pyramid)”) on Table 3.1.

Listing 3.3: Code snippet to perform image resizing using GPU texture memory

```
1 // Texture memory to stored input image.
2 // Some configurations (e.g., setting the flag for automatic
  ↳ interpolation) are performed on the host (CPU) side code
3 texture<float, cudaTextureType2DLayered,
  ↳ cudaReadModeElementType> org_image;
4
5 // Lookup table to store accumulated image size (i.e.,
  ↳ accumulated memory amount)
6 texture<int, cudaTextureType1D, cudaReadModeElementType>
  ↳ image_idx_incrementer;
7
8 // Function body
```

```

9  __global__
10 void resize (
11 int src_height, int src_width, float *dst_top,
12 int dst_height, int dst_width, float hfactor,
13 float wfactor, int level) {
14     // Specify indices of this CUDA thread
15     int dst_x   = blockIdx.x*blockDim.x + threadIdx.x;
16     int dst_y   = blockIdx.y*blockDim.y + threadIdx.y;
17     int channel = blockIdx.z;
18
19     // Get destination where this CUDA thread should treat
20     float *dst = dst_top + tex1Dfetch(image_idx_incrementer,
21     ↪ level)
22     + channel * dst_height * dst_width;
23
24     // Get source image location where to see
25     float src_x_decimal = wfactor * dst_x + 0.5f;
26     float src_y_decimal = hfactor * dst_y + 0.5f;
27
28     // Execute interpolation using the dedicated fetch operation
29     // with desired index and store the result
30     if (dst_x < dst_width && dst_y < dst_height) {
31         dst[dst_y*dst_width + dst_x] =
32         ↪ (float)tex2DLayered(org_image, src_x_decimal,
33         ↪ src_y_decimal, channel);
34     }
35 }

```

### 3.1.3 Results of DPM acceleration using GPUs

To quantify the performance of the GPU-based DPM implementation, the effects of the following factors were evaluated: (i) type of GPU, (ii) parallelization scheme, (iii) texture and shared memories, (iv) image sizes, (v) object classes, (vi) number of threads, (vii) resizing algorithms, (viii) memory copies between the CPU and GPU, and (ix) detection accuracy after applying the GPU-based implementations.

#### Experimental Setup

An Intel Xeon CPU, E5-2687W v2, with eight cores and operating at 3.40 GHz was used as the host processor. Multithreaded programs on the CPU were implemented using the POSIX Thread Library (*pthread*). The DPM was implemented using CUDA on various NVIDIA GPUs, including (i) GeForce GTX 560 Ti, (ii) GeForce GTX 680, (iii) GeForce GTX TITAN, (iv) GeForce GTX TITAN Black, and (v) Tesla K20Xm. These GPUs were chosen from the popular lines of NVIDIA products; therefore, they reflect state-of-the-art performances of the proposed DPM implementation schemes. The CUDA code was compiled using the *nvcc* compiler with the common options: `nvcc -cubin -Xptxas -v --maxrregcount 32`. In addition, the performances of different GPU architectures with different specifications within the same GPU architecture can be compared. Since the newer GPUs provide greater performances, it is expected that future GPUs may allow further improvements to DPM performance.

The code for the DPM and its target models used herein can be obtained from the author's website<sup>1</sup>. Since the original code was written in MATLAB, it was re-implemented in C++ and extended to CUDA for the GPU-based DPM implementation.

The car and pedestrian models were used in the evaluations. The numbers of resized HOG feature-quantity images ( $N$ ) and filters ( $M$ ) were fixed at  $N = 25$  and  $M = 12$ , in accordance with the original code. In the evaluation of the proposed GPU-based DPM implementation, the sizes of the input images and types of filters were changed to assess the performance scalability with respect to input data, instead of changing the values directly.

#### Effects of the Types of GPUs

The key specifications of the GPUs used in the experiments are listed in Table 3.2. Fig. 3.10 and Table 3.3 show the execution times of the DPM on various GPUs compared with those on CPUs for a car model from VOC 2007 [24] as the object class. Eleven VGA images ( $640 \times 480$  pixels) are used as the input data, and the average execution times are derived.

---

<sup>1</sup><http://people.cs.uchicago.edu/~rbg/latent-release5/>

Table 3.2: Key specifications of GPUs used in the evaluations.

	Released	# of CUDA cores	Base clock [MHz]	Memory band- width [GB/s]	Architecture
GTX 560 Ti	Jan. 2011	384	823	128.3	Fermi
GTX 680	Mar. 2012	1536	1006	192.3	Kepler
Tesla K20Xm	Nov. 2012	2688	732	249.6	Kepler
GTX TITAN	Feb. 2013	2688	836	288.4	Kepler
GTX TITAN Black	Feb. 2014	2880	889	336.0	Kepler

The data for “Xeon (single)” denotes the original sequential execution of the DPM, while the “Xeon (multi)” data refers to the multithreaded DPM implementation on the same CPU, where only the HOG pyramid and score computations are parallelized to maximize the DPM performance. This indicates that parallelizing the computations of the distance transforms and score summations significantly decreases the performance owing to multicore characteristics. The “Xeon (SSE)” data denotes the execution time for the same code as “Xeon (multi)”, but the compilation was achieved using an Intel C++ Compiler with `-O3 -fast` options to accelerate CPU-based computations. The other items represent the results from the corresponding GPUs, where hybrid parallelization was applied for score computations and HOG pyramid construction, which was experimentally determined to be the best combination in terms of performance.

The above results demonstrate the significant advantages of GPUs in terms of execution times. The execution time of the DPM for C++ implementation was 1541ms and that achieved by multithreaded imple-



### 3.1. Accelerating traditional object detection algorithm using GPUs

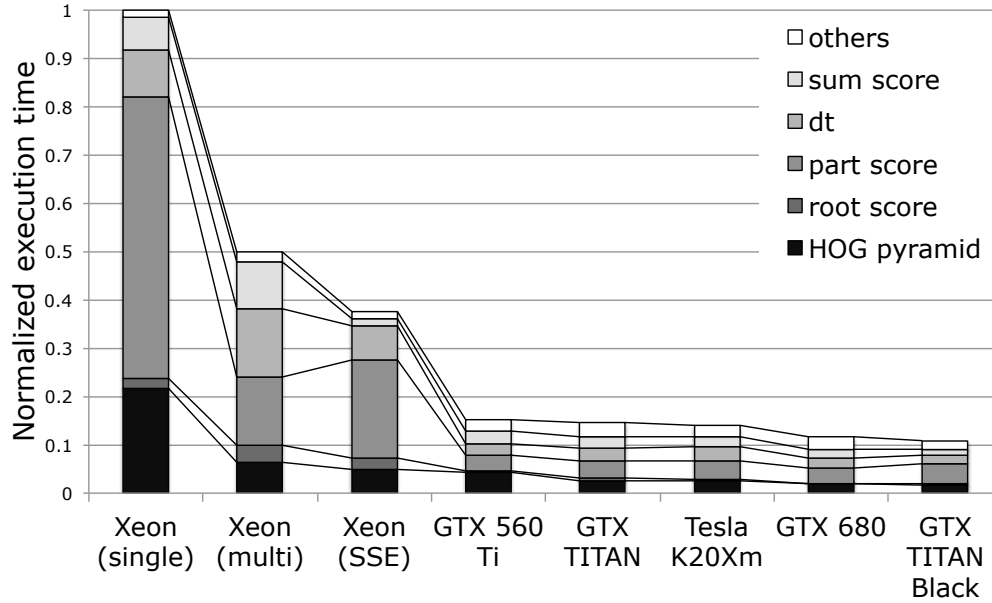


Figure 3.10: Execution times of the DPM program on GPUs and CPUs.

mentation was also at most 771ms. In contrast, using GPUs allows the DPM to perform at best in 178ms, which is an approximately 8.6-fold improvement over the C++ implementation and 4.3-fold improvement over the multithreaded implementation. Additionally, the execution time of the DPM for vector parallelization (SSE) using an Intel C++ Compiler is 578ms, for which the proposed GPU implementation is still approximately 3.2 times faster. Comparing GPUs in Table 3.2, GTX TITAN Black achieved the best performance. The recent trend of GPUs released from NVIDIA is to have more CUDA cores, lower base clock, and wider memory bandwidth to process many numbers of single instruction for multiple data efficiently. Obeying this trend, GTX TITAN Black, which is the latest GPU used in this evaluation, has the most number of CUDA cores and widest memory bandwidth. Although the most time-consuming parts of the DPM are computationally intensive, some of them contain considerable data access operations. Hence, it is assumed that both of high computational and memory access efficiency of GTX TITAN Black result in its performance.

As shown in Fig. 3.10 and Table 3.3, compared with the sequential implementation of “Xeon (single)”, the functions have longer execution times

Table 3.3: Actual execution times corresponding to Fig. 3.10 (in milliseconds).

	Xeon (single)	Xeon (multi)	Xeon (SSE)	GTX 560 Ti	GTX TI- TAN	Tesla K20Xm	GTX 680	GTX TI- TAN Black
others	24.83	32.85	21.54	39.26	36.48	46.38	36.75	37.67
sum score	101.32	151.21	22.30	33.83	42.84	36.65	31.51	29.80
dt	149.21	215.46	108.88	43.94	35.32	40.08	42.63	32.56
part score	900.30	221.21	315.65	142.46	49.38	56.58	60.42	46.34
root score	31.48	51.42	32.24	8.37	6.83	5.29	4.17	2.86
HOG pyra- mid	334.67	99.56	77.79	82.82	65.43	41.49	40.29	29.02
total	1541.82	771.70	578.40	350.67	236.27	226.46	215.78	178.26

in the “Xeon (multi)” multithreaded implementation. This is attributed to the fact that multithreaded implementation incurs an overhead for creation of new CPU threads. Note that the number of CPU threads required for `root score` is fewer than that required for `part score`. Similarly, a specific function such as `part score` of vector parallelization, labeled “Xeon (SSE)”, is slowed compared with that of the multithreaded implementation. In this case, it is assumed that a lot of instructions are offloaded to the SSE unit in `part score` because of the large computational requirement; therefore, the SSE unit is saturated and the overhead of parallelization exceeds the acceleration effect brought by parallelization. The same phenomenon may also occur on the GPU; if too many threads are created on the GPU, the computational performance would be significantly degraded.

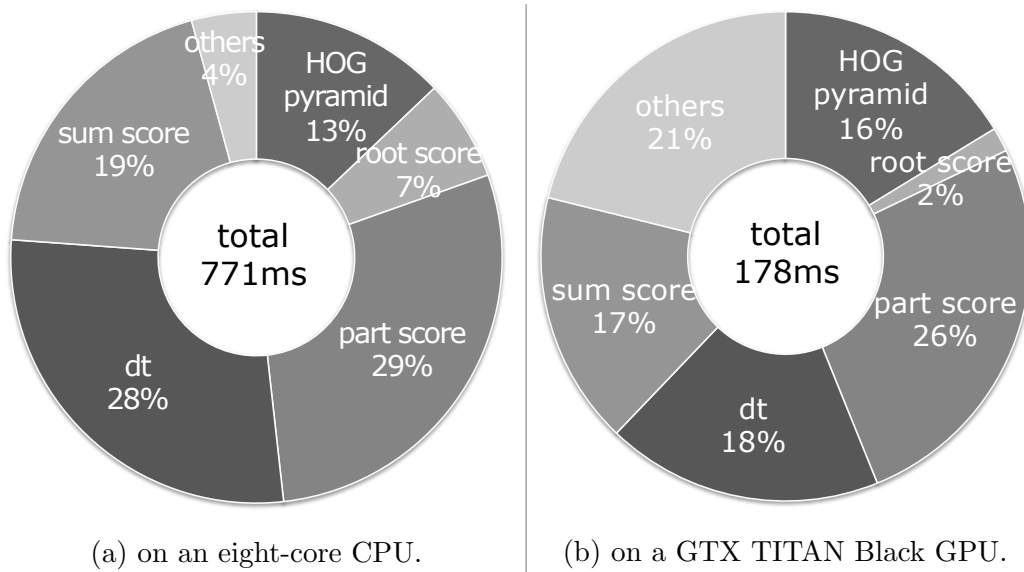


Figure 3.11: Breakdown of the execution times on two platforms.

The breakdowns of the execution times for the multithreaded and GPU-based implementations are shown in Fig. 3.11a and 3.11b, respectively. Note that only the results for the GTX TITAN Black are presented here owing to space constraints. Comparing these results with those in Fig. 3.2 clearly shows that the percentages of the computationally intensive data-parallel parts (HOG pyramid, root/part scores, dt, and sum score) are significantly lower for GPU implementation than the multicore case, which characterizes the performance improvements with the GPU for DPM.

A previous study [38] demonstrated that a multithreaded DPM implementation would yield a shorter execution time than GPU-based DPM implementation for one of the bottlenecks, namely the HOG pyramid, because hybrid parallelization was not applied. This finding indicates that numerous idle threads were generated on the GPU, thereby wasting GPU resources; hence, the performance of the GPU code was not optimal. On the other hand, the results of the present work prove that hybrid parallelization, based on a balancing point between the CPU and GPU threads, increases the speed of HOG pyramid construction by approximately four times than that with multithreaded implementation. This confirms that

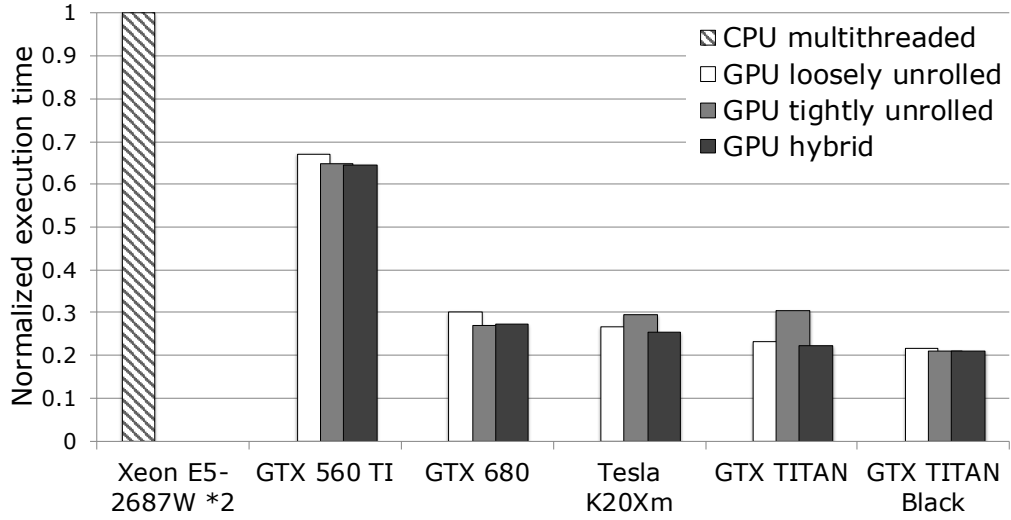


Figure 3.12: Effects of the parallelization schemes on part score computations.

the proposed hybrid parallelization successfully improves GPU-based DPM implementation performance.

### Effects of Parallelization Schemes

The three proposed parallelization schemes are compared as follows. In particular, the part score and image resizing computations, which are two of the major procedures under DPM, are focused on. Note that image resizing cannot utilize hybrid parallelization as the computations do not involve the  $z$ -plane, whereas the computations of the part scores allow comparisons among the three schemes.

Execution times for the computation of scores with the proposed GPU parallelization schemes are compared with those for the CPU multithreaded implementation in Fig. 3.12.

With regard to the part score computations, while the algorithm needs to read data from the HOG images and model filters to compute the pixelwise similarities, the computations of the similarities themselves can be parallelized, as discussed in Section 3.1.2. Hence, such computations involve both numerical calculations and data accesses. Among the proposed GPU

### 3.1. Accelerating traditional object detection algorithm using GPUs

---

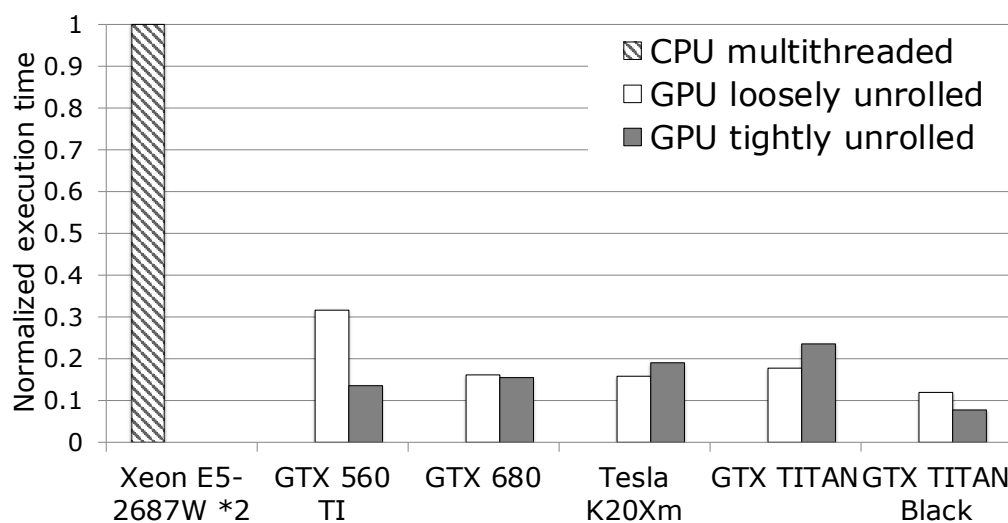


Figure 3.13: Effects of different parallelization schemes on image resizing.

schemes, hybrid parallelization outperforms the others, except in the case of the GTX 680, although this is considered to be within the acceptable range of errors. A comparison of the loosely and tightly unrolled parallelization schemes indicates that neither dominates the other, meaning that fewer idle threads do not necessarily improve computational performance. Hence, the hybrid parallelization scheme is recommended for accelerating DPM computations, which creates  $N$  CPU threads to launch  $M$  GPU CUDA threads, where  $N$  and  $M$  are the numbers of resized images and model filters, respectively, as described in Fig. 3.7. Note that  $N$  and  $M$  are determined at the time of model training (or learning) performed before detection; thus, these are fixed values on detection, which is the acceleration target of this work.

The execution times for image resizing, which is part of the HOG pyramid Construction step, by loosely and tightly unrolled parallelization are compared with those of the CPU multithreaded implementation in Fig. 3.13. Note that hybrid parallelization is not applied to image resizing as the computational blocks do not have a  $z$  dimension; that is, each resized image is represented in the  $x$ - $y$  plane.

The image resizing procedures first access multiple pixel values in the input image and then compute the corresponding pixel values. As

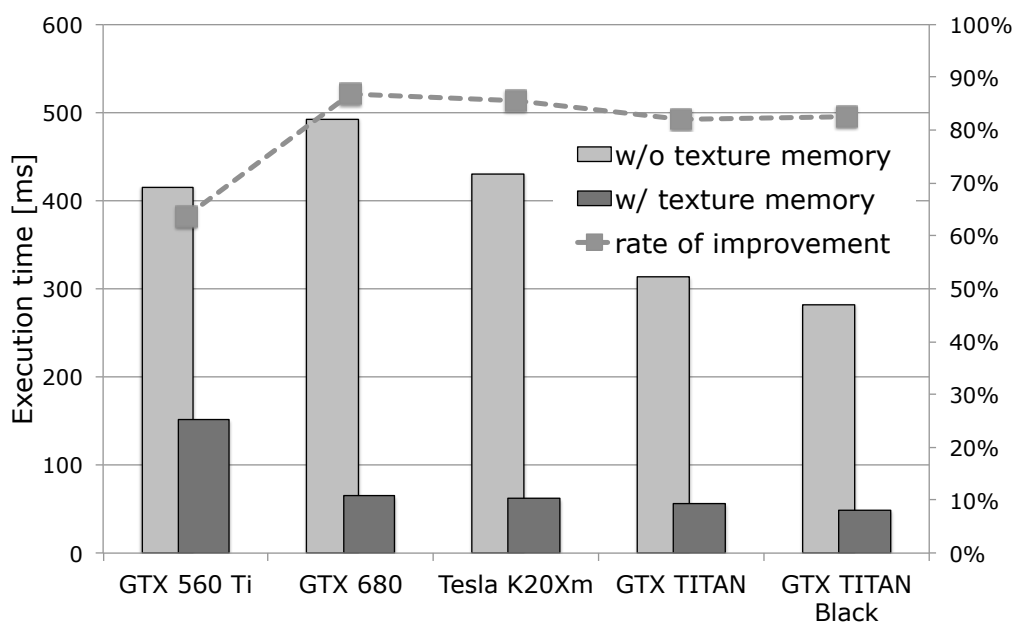


Figure 3.14: Effects of texture memory on part score computations.

discussed in Section 1.1.2, the pixel values can be automatically computed by hardware using the texture and shared memories. Therefore, image resizing involves intensive data accesses rather than numerical computations. This characterization of image resizing, as opposed to part-scores computation, explains why the parallelization schemes may not really matter from the viewpoint of execution time.

### Effects of Texture and Shared Memories

The effects of texture memory on the part score computations and HOG pyramid construction are demonstrated in Figs. 3.14 and 3.15, respectively. Note that the texture memory is used for read-only data; as discussed in Section 1.1.2, although the texture memory is constrained in terms of only being read from the CUDA threads, its access speed is greater than that of global memory. Thus, if only data referencing is needed, the execution times of the GPU kernels can be reduced by storing data in the texture memory.

In the computation of HOG features, the edge intensities and edge gradients are computed from each pixel of the resized image to generate

### 3.1. Accelerating traditional object detection algorithm using GPUs

---

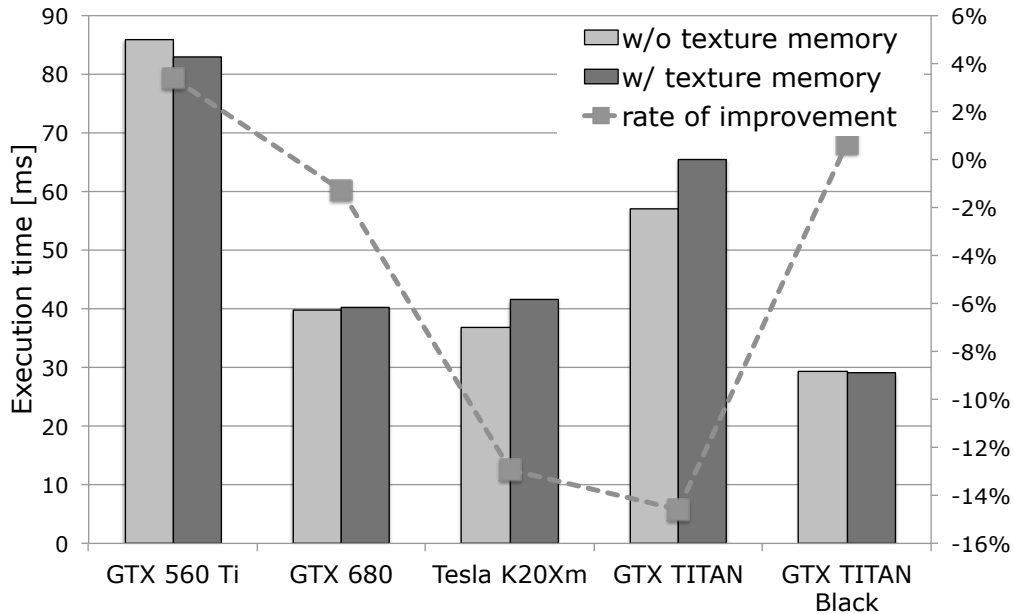


Figure 3.15: Effects of texture memory on HOG feature computations.

a histogram. This task includes normalization of the histograms and other processes, implying more numerical computations than data accesses, while image resizing is by itself highly data intensive. The part score computations involve both numerical calculations and data accesses, as mentioned above. Therefore, the execution time for part score computation is significantly reduced when using texture memory, whereas that for HOG feature computation is not reduced much owing to a small number of memory accesses. Note that in some cases, the HOG feature computation performance is degraded when using texture memory; this is because the overhead incurred by referencing texture memory could outweigh the number of data access reductions.

In CUDA, the texture memory is used as a read-only cache for the global memory, while the shared memory is used as temporary memory. To complement the effects of memory usage, a variant of the GPU-based DPM implementation using shared memory was evaluated. With the exception of HOG pyramid construction, the DPM does not require shared memory.

The effects of shared memory on HOG pyramid construction are shown

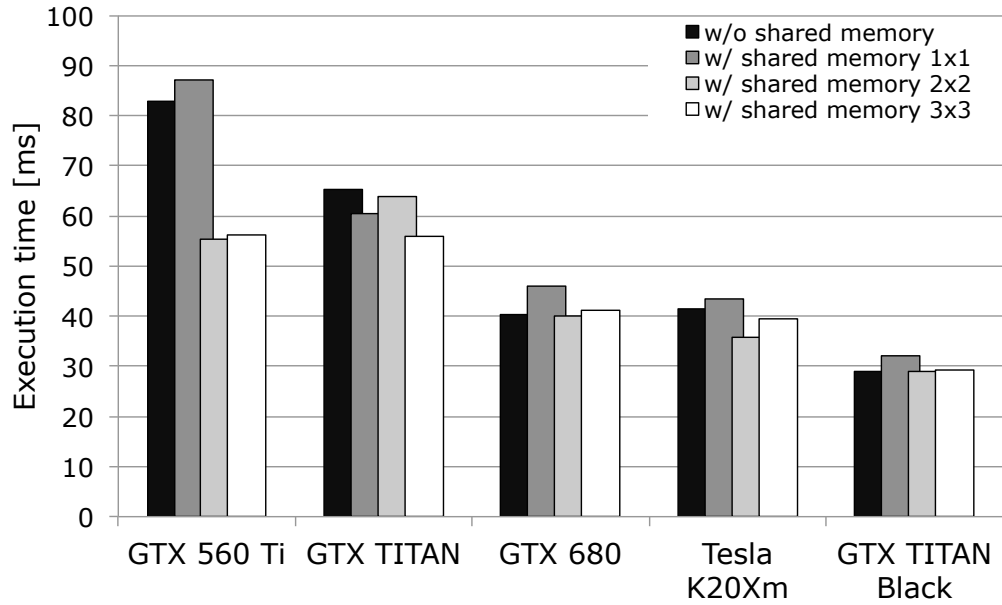


Figure 3.16: Effects of shared memory on HOG feature computations.

in Fig. 3.16. The execution times before applying shared memory are labeled “w/o shared memory”, while those using shared memory are labeled as “w/ shared memory  $k \times k$ ”, where  $k \times k$  denotes the number of cells in the image computed by a block. A higher number of cells per block requires a higher number of threads per block, thereby increasing the efficiency of shared memory usage.

Except for the GTX 560 Ti, the introduction of shared memory did not contribute to considerable performance improvement. According to the CUDA Profiler, the HOG pyramid construction is marked as “Kernel Performance is Bound By Compute”, i.e., the execution time for HOG pyramid construction is not dominated by memory accesses. The impact of shared memory, therefore, was not significant. In the case of the GTX 560 Ti, however, some effects were observed because this GPU is not designed to have a high memory bandwidth, meaning that shared memory may resolve this bottleneck.



### 3.1. Accelerating traditional object detection algorithm using GPUs

---

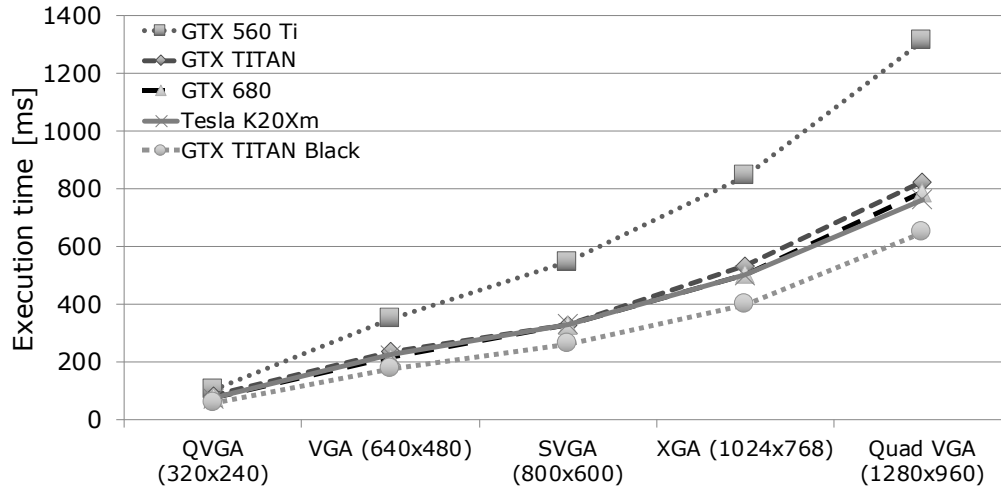


Figure 3.17: Effects of input image resolution on various GPUs.

#### Effects of Image Size

The execution times of the accelerated DPM on various GPUs for changes in the input image resolution within the range of  $\{320 \times 240, 640 \times 480, 800 \times 600, 1024 \times 768, \text{ and } 1280 \times 960\}$  are shown in Fig. 3.17.

For an input image resolution of  $1280 \times 960$  pixels, the execution time achieved by the GTX 560 Ti, a ten-year-old GPU, is approximately 1314 ms, whereas that on the GTX TITAN Black, the best performer, is approximately 651 ms. These results represent a roughly two-fold difference in the execution times of the GPUs. Among the GPUs considered, only the GTX 560 Ti is based on the Fermi architecture [39], while the others are based on the Kepler architecture [40]. Compared to the Fermi architecture, the Kepler architecture has superior performance for multiple CUDA streams owing to its improved technology, including Hyper-Q [40]. Therefore, it is not surprising that advances in GPU architectures have enabled a two-fold improvement in computing performance over the past decade.

The execution times corresponding to various image resolutions shown in Fig. 3.17 are depicted in Fig. 3.18. The proportion of execution time required for the GPU kernel is approximately 48% for an input image resolution of  $320 \times 240$  pixels on the GTX 560 Ti, 67% for  $640 \times 480$  pixels, and more than 70% for higher resolutions. The exception to this trend is the GTX

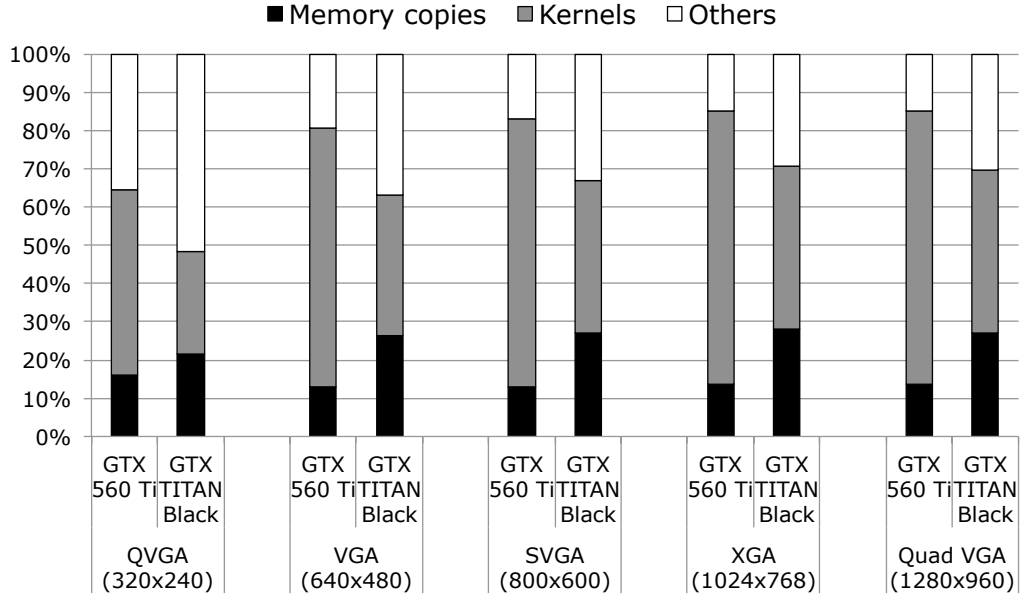


Figure 3.18: Breakdown of the execution times for different image resolutions on the worst- and best-performing GPUs.

TITAN Black, whose proportion of GPU kernels is approximately 26% for an input image resolution of  $320 \times 240$  pixels and 40% for higher resolutions. Regardless of the GPU type, the ratio of execution time for memory copies (data copies between the GPU and CPU and within the GPU) to the total execution time is lower than the other processes. Thus, for the tested workload (accelerated DPM), a performance bottleneck occurs owing to the numerical computation, and increasing the speed of numerical computations may effectively improve the performance of the DPM.

The execution times of the proposed accelerated DPM for RGB color and grayscale images are compared in Fig. 3.19. An input image resolution of  $640 \times 480$  pixels was used with the various GPUs.

It was anticipated that the execution time with the grayscale image would be shortened because of reduction in the amount of data. However, the GTX TITAN Black, which exhibits the largest improvement, achieves an execution time reduction of only approximately 14 ms for the grayscale images, i.e., from 178 ms to 164 ms. This negligible reduction may be attributed to the fact that the HOG features are not dominated by color information, as

### 3.1. Accelerating traditional object detection algorithm using GPUs

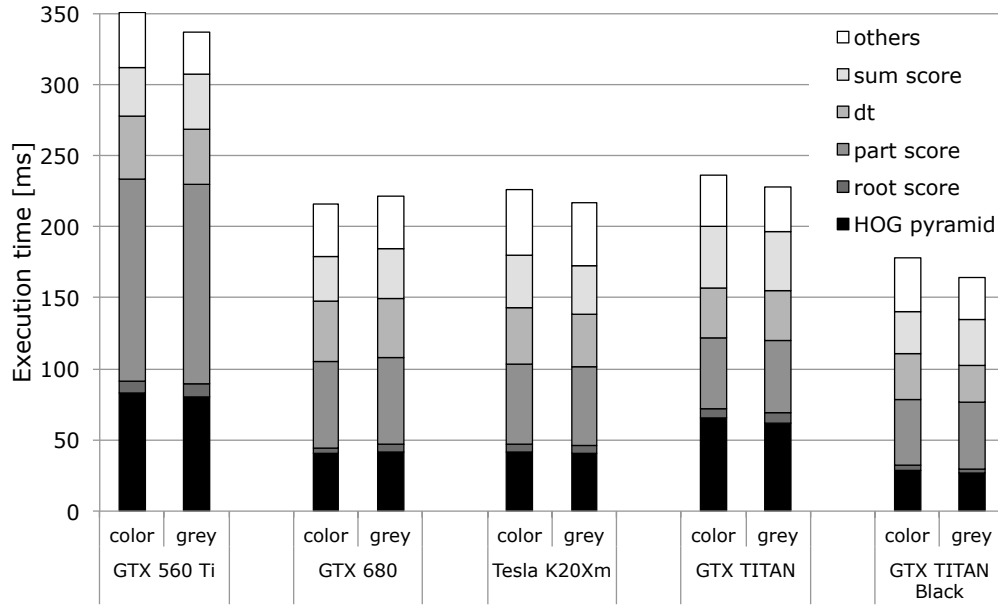


Figure 3.19: Effects of color and grayscale images.

Table 3.4: File sizes of model filters for pedestrians and vehicles.

	Root Filter	Part Filter
Vehicle	48K	206K
Pedestrian	39K	165K

described in Section 3.1.1; furthermore, the differences in the numbers of colors in the input images are not sufficient to significantly influence the execution times between RGB and grayscale images.

#### Effects of Object Classes

The execution times of the accelerated DPM when the object detection targets, such as vehicles and pedestrians, differ are compared in Fig. 3.20. An input image resolution of  $640 \times 480$  pixels was used in both cases.

Regardless of the GPU type, the execution time required to recognize pedestrians is shorter than that for vehicles. In this experimental evaluation, the DPM program stores the model filters for the corresponding object classes

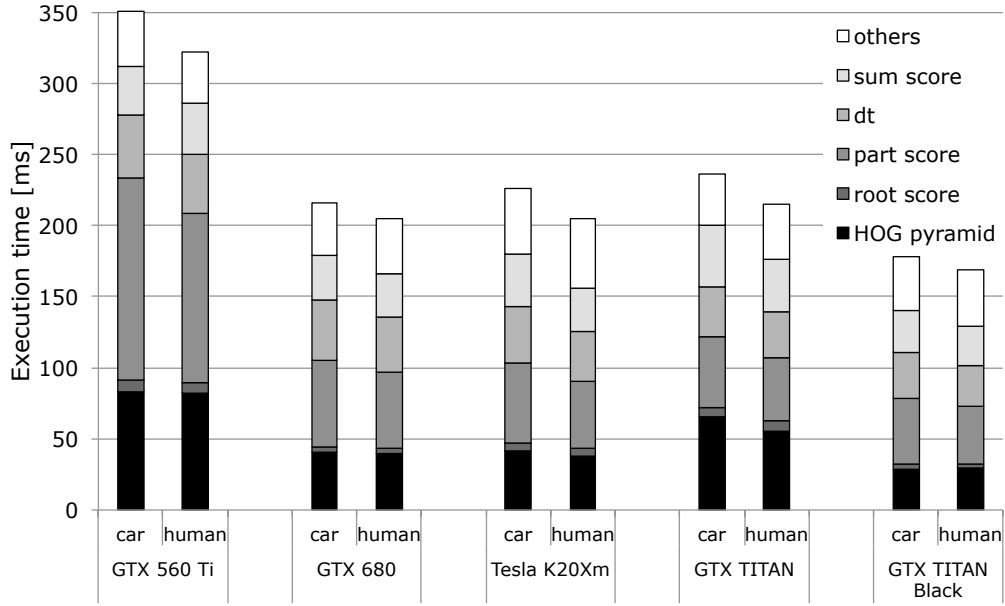


Figure 3.20: Effects of object classes (vehicle and pedestrian).

in csv format, and these files are read during program execution. As listed in Table 3.4, the file sizes are smaller for the pedestrian models than those for the vehicle models. Although the DPM algorithm uses two filters, i.e., root and part filters, the same is true for both filters. It is hypothesized that these differences in model sizes are responsible for the differences in execution times of the score computations, which consequently reflect upon the overall execution time.

### Effects of the Number of Threads

For all GPUs, the execution times for all scenarios are least when the maximum number of threads per block is either 128 or 256. The GPUs allow numerical computations and memory accesses to overlap by dispatching the active threads whenever ready on each core, thus mitigating the latencies in memory accesses. When the GPU kernels are executed, GPU resources are allocated to each block, as noted in Section 1.1.2; hence, when the number of threads per block is low, the memory latencies cannot be hidden because the corresponding execution times are long, as observed from Figs. 3.22 and 3.21.

### 3.1. Accelerating traditional object detection algorithm using GPUs

---

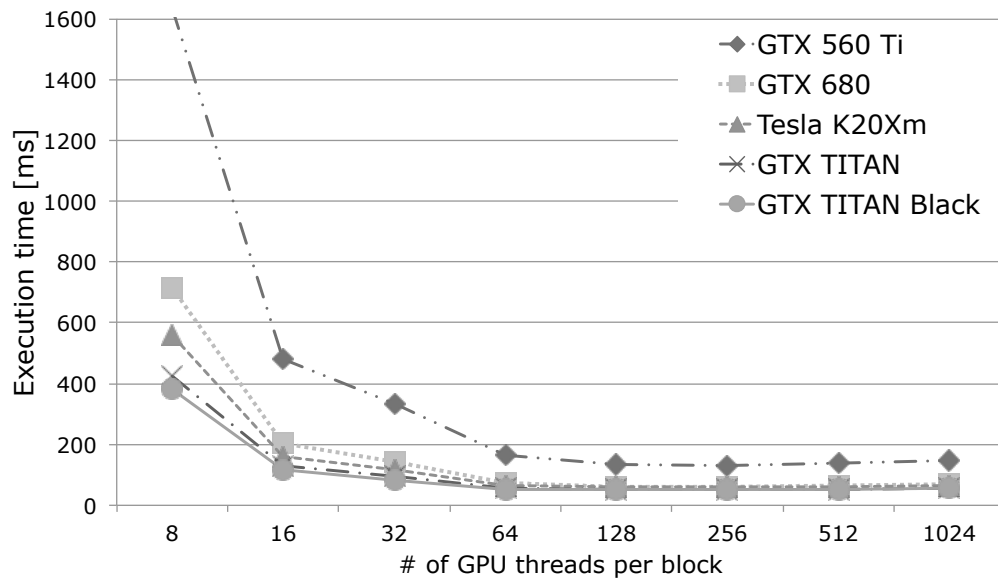


Figure 3.21: Effects of maximum number of threads per block in loosely unrolled parallelization.

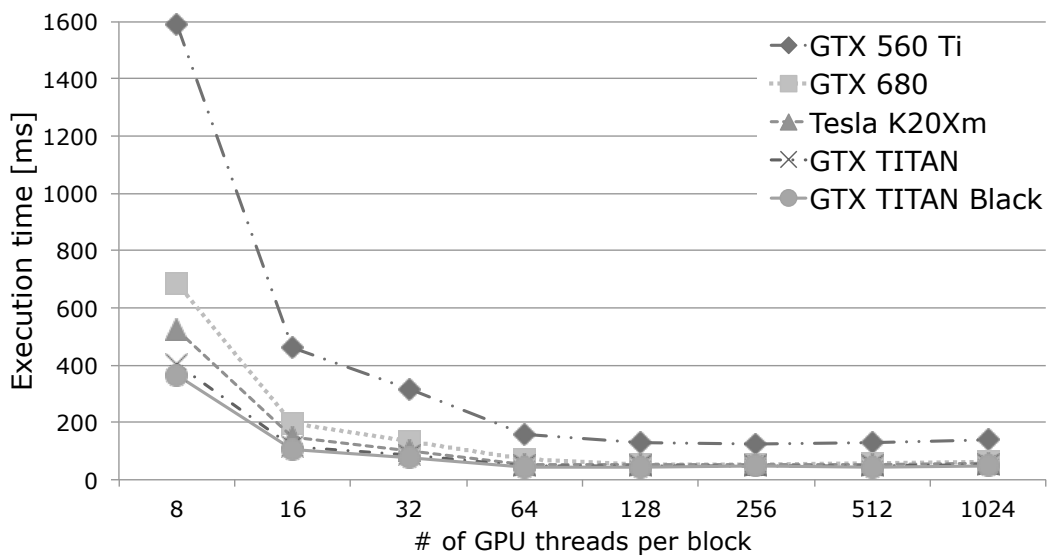


Figure 3.22: Effects of maximum number of threads per block in tightly unrolled parallelization.

Additionally, it is inferred that if the maximum number of threads per block exceeds some optimal number, the computation cost for each thread exceeds

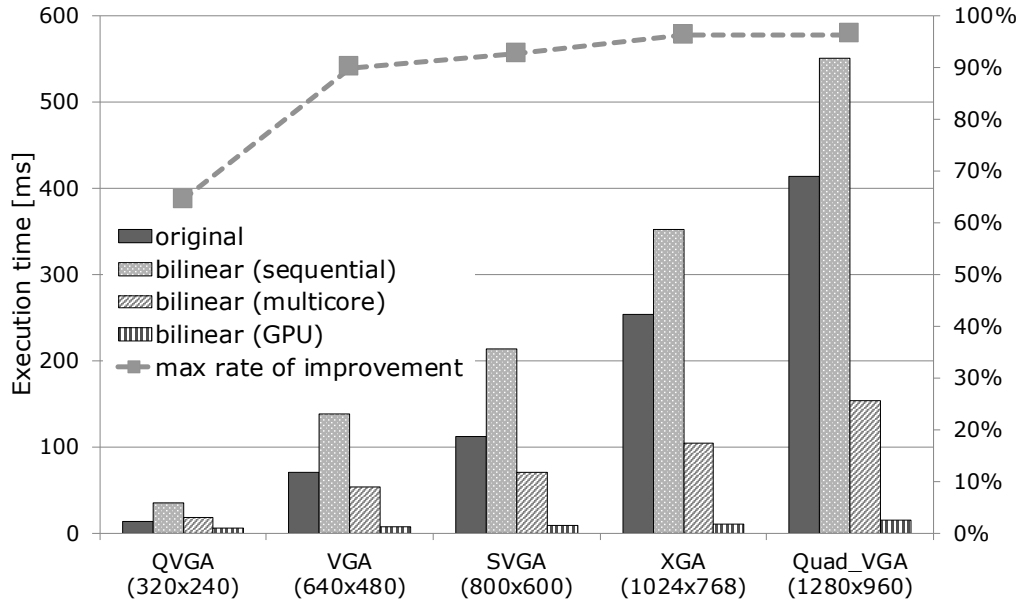


Figure 3.23: Effects of resizing algorithms.

the latency of memory access, creating another bottleneck that increases the execution time.

### Effects of Resizing Algorithms

The resizing algorithms used for the input images to generate the HOG pyramids are compared in Fig. 3.23. The following algorithms were implemented: (1) algorithm implemented in the original DPM [35] (labeled as “original”), (2) bilinear interpolation algorithm written in the C language (labeled as “bilinear (sequential)”), (3) parallelized bilinear interpolation algorithm using pthreads (labeled as “bilinear (multicore)”), and (4) parallelized bilinear interpolation algorithm using CUDA (labeled as “bilinear (GPU)”).

It is interesting to note that the original algorithm [35] is faster than the bilinear interpolation algorithm if implemented in C language, i.e., using a single-thread implementation. However, the original algorithm does not provide parallelism (owing to implementation constraints), so it cannot use GPUs effectively. On the other hand, the bilinear interpolation algorithm is highly parallelizable and can be implemented using pthreads and CUDA. When the input image size was Quad VGA, the execution time of

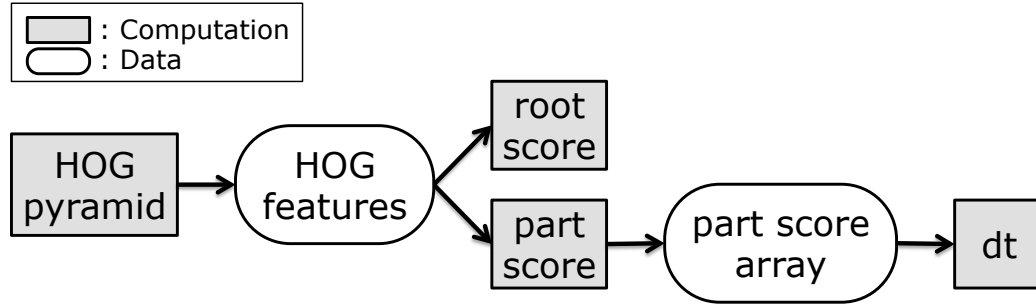


Figure 3.24: Data flow in DPM.

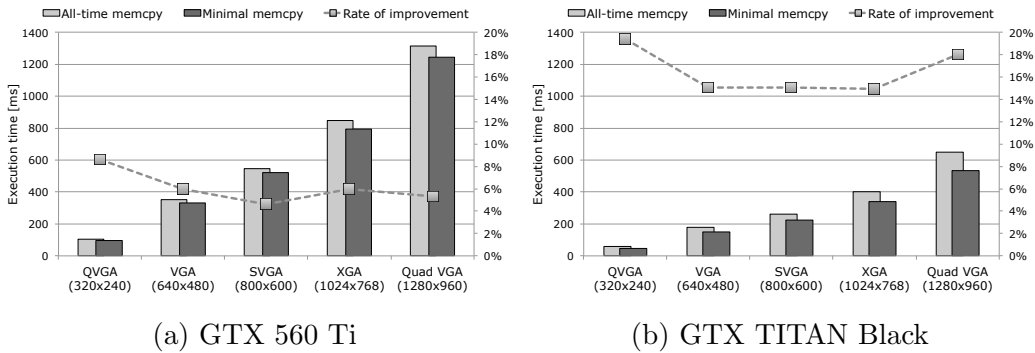


Figure 3.25: Effects of memory copies between the host and device memories for the GTX 560 Ti (left) and GTX TITAN Black (right).

the GPU-accelerated bilinear interpolation algorithm was approximately 30 times less than that of the original algorithm; the reason for this improvement is largely attributed to the fact that data parallelism and the simplicity of bilinear interpolation are suitable for GPU implementation, while a certain performance benefit is gained from the hardware-interpolating function of the texture memory.

### Effects of Memory Copies

In a DPM computation such as that shown in Fig. 3.24, two pieces of data are exchanged between the computing blocks. Since a GPU is used to implement each computing block as an individual GPU kernel, these data must be managed so that the output of one kernel can be input into the next kernel because each GPU kernel is self-contained (i.e., they do not communicate

with the other kernels). The most straightforward DPM implementation is to copy the data between the host and device memories, as is typically used in CUDA programming; that is, the output of one kernel is copied to the host memory accessible to the CPU and is copied again to the device memory accessible to the GPU so as to enable reading by the next kernel. Another scheme for implementing DPM is to share the pointers to these data among the GPU kernels. This scheme is preferred because the overhead from memory copies can be reduced.

The effects of the memory copies on the execution times of the accelerated DPM are shown in Fig. 3.25. It was demonstrated using two GPUs, i.e., the GTX 560 Ti (older Fermi) and GTX TITAN Black (relatively newer Kepler), whether the different characteristics of these GPUs could be observed through the changes in their architectures. The rate of improvement achieved by reduction of the memory copies on the GTX 560 Ti is up to 8.6%, whereas that on the GTX TITAN Black is approximately 20% under identical conditions. This characterization can be reasoned as follows. As seen in Fig. 3.18, the GTX 560 Ti spends the most time executing GPU kernels rather than memory copies, while the GTX TITAN Black reduces this time to almost an equal percentage as that of the memory copies (owing to its improved compute cores). The overhead for the memory copies thus contributes more to the total performance of the GPU than the total execution time on the GTX TITAN Black. This observation indicates that the overhead for memory copies could become more significant as the performances of the compute cores improve with GPU architecture.

### **Effects of GPU Implementation on the Detection Rates**

To verify the accuracy of the proposed GPU implementation, the detection rates between CPU and GPU implementations of the DPM are compared as follows.

The VOC 2007 [24] dataset is used with the following experiments.

1. Extract 1,000 random images allowing duplicates from the images for car model training in the VOC 2007 dataset.
2. Process the extracted images by CPU implementation of DPM and



### 3.1. Accelerating traditional object detection algorithm using GPUs

---

save the image indices and number of detected objects for each image.

3. Repeat the same process for GPU implementation of DPM.
4. Compare the results.

Comparing the computation results for the CPU and GPU implementations reveals that the results match for 964 of the 1,000 extracted images, while 36 images do not match, although the differences in the mismatched results are trivial. A possible reason for this mismatch may be the differences in the precisions of the floating-point calculations between the CPU and GPU. Although recent GPUs adopt the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1875 [41]), in some cases, the results of floating-point calculations of the GPU do not match with those of the CPU owing to the uniquely expanded calculation function of GPUs [42]. Since the DPM estimates the target object on the basis of a threshold, even sparse errors can affect the detection rates if they occur around the threshold. Nonetheless, the rounded values of the computation results obtained by both the CPU and GPU implementations are the same since errors can occur only in the case of low-order bits, and approximately 96% of the detection results obtained by the two algorithms are matched. It can therefore be concluded that the proposed GPU-based DPM implementation is superior in terms of the execution time.

#### 3.1.4 Results of GPU-accelerated traditional object detection

This section presented the GPU-based DPM implementation schemes to validate GPU applicability to one of the traditional pattern recognition task. The performance improvements using GPUs are illustrated here, along with their detailed quantitative evaluations using different setups for DPM acceleration on a GPU. From the analysis, the DPM comprises five major computing blocks, namely the *HOG pyramid*, *root score*, *part score*, *dt*, and *sum score*, that account for about 98% of the total computations, all of which exhibit loop processing.

Multiple implementation schemes were provided here based on the massively parallel compute threads using event streams and texture memory. The best scenario of GPU implementation achieved a performance improvement of  $8.6\times$  over a high-end CPU implementation. This achievement is significant because the algorithm remained unchanged but significantly improved in speed, whereas prior works could only speed up the algorithm at the expense of decreased detection rates.

Although a large performance improvement was achieved, the results also imply that there is a limit to accelerating the computational speeds of traditional pattern recognition algorithms with GPUs. The best scenario of GPU implementation required a total execution time of approximately 179 ms ( $\approx 5.6$  frames per second (fps)). At this processing rate, frame drops occur when images are input at the rates of commonly used sensors for autonomous driving (typically at least over 10 fps). Moreover, the perception performances are not sufficient to perceive the complex surroundings and circumstances in driving environments. These factors prevent adopting the DPM in the high-level autonomous driving modules.

## 3.2 Paradigm shift forming current mainstream

In computer vision, the term *features* often refers to information that describes the contents of an image. For many years, image processing techniques, including object detection, relied on hand-crafted features, which were designed by humans to fit specific purposes. To improve the processing speed and accuracy, hand-crafted features such as SIFT [14], Haar-like [15], and HOG [16] have been proposed and have significantly contributed to the growth of the field. However, the growth of image processing performance using hand-crafted features has gradually stagnated. Since around 2012, with the improvements in the computational capabilities of computers and the availability of large-scale data, neural network techniques have been revived for image processing because these techniques can generate more complex features from large amounts of training data and often surpass

### 3.2. Paradigm shift forming current mainstream

---

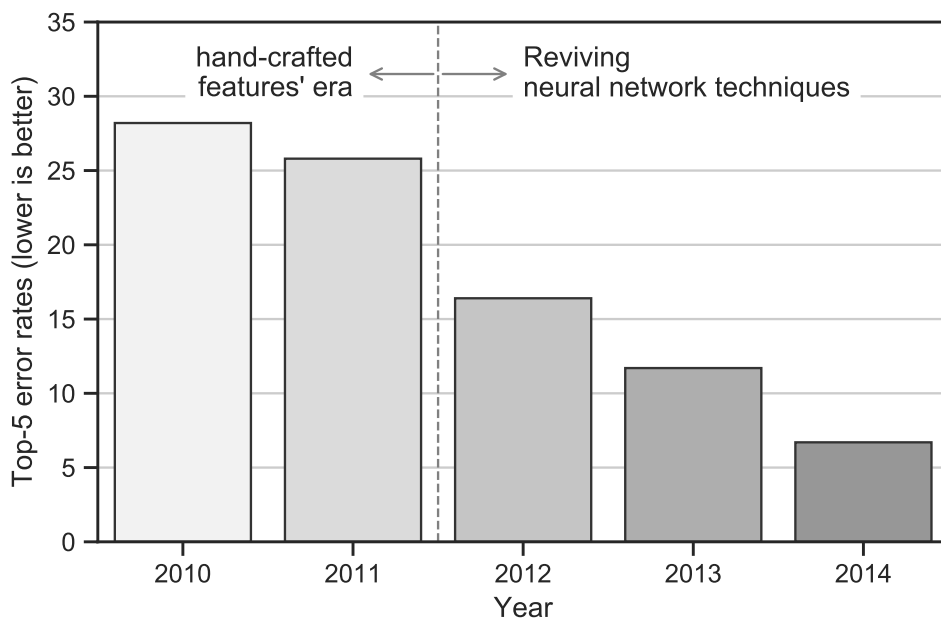


Figure 3.26: Winning scores of the ILSVRC classification tasks; the values are cited from [23].

human recognition capabilities. This trend can be easily seen in the error rate transitions of object category classifications in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [23], which is a major benchmark for object category classification and detection containing hundreds of object categories and millions of images. Fig. 3.26 shows the winning scores (Top-5 classification error rates: the rate for which the algorithm's predictions with the top-five confidence scores do not contain true classes) of the ILSVRC for five consecutive years. In the ILSVRC2012 competition, the algorithm that utilized a large-scale deep CNN [4] won by a large margin. This victory was a turning point for deep-learning techniques, which have attracted attention since then as possible solutions to a wide range of problems.

These deep-learning tendencies have also affected field of object detection. Table 3.5 shows a comparison of the detection accuracies and processing speeds of some representative object detection algorithms. The object detection algorithms using deep-learning techniques can be roughly categorized into two groups, namely *two-stage* and *one-stage* detectors [44]. The faster

Table 3.5: Performance comparison of detection algorithms; the detection mean average precisions (mAPs) are cited from or calculated using the reference papers’ values for PASCAL VOC2007 data. The fps values, except for DPM, are cited from [28]. For the DPM, the fps is obtained from the best scenario in Table 3.3. Note that the fps values are only given for reference since the execution times may vary according to various factors, including the implementation-framework used and hardware settings.

	DPM [12]	Faster-RCNN [43]	YOLO [27]	SSD [28]
	traditional pattern recognition base	deep-learning base		
Proposed	2008	2015	2016	2016
detection mAP	21.3	73.2	66.4	74.3
fps (for reference)	5.6	7	21	46

---

### 3.2. Paradigm shift forming current mainstream

---

region-based CNN (Faster-RCNN) [43] is a representative two-stage detector that typically achieves high localization and object recognition accuracies. On the other hand, YOLO [27] and SSD [28] are grouped under one-stage detectors, whose typical distinctive feature is the high inference speed. Comparisons with and without deep-learning techniques show that the detection accuracy (“detection mAP” in Table 3.5) is significantly improved by introducing deep learning regardless of one-/two-stage detectors. Moreover, these three deep-learning-based algorithms are faster than GPU-accelerated DPM, especially one-stage detectors that achieve real-time inference speeds (typically 10–30 fps). As noted in Section 1.1, in high-level autonomous driving systems, the perception tasks have to be processed with small delays as well as high accuracies such that the subsequent modules can decide appropriate driving behaviors. From the perspective of detection accuracy and speed, it can be concluded that applying deep-learning techniques is inevitable for implementing the perception modules of high-level autonomous driving systems.

## Chapter 4

# Traffic light recognition using High Definition Map Features

### 4.1 Motivation

It is common knowledge that vehicle-to-roadside-infrastructure (V2I) and vehicle-to-vehicle (V2V) communications help improve traffic safety and autonomous driving functionalities. These systems share information such as traffic light states and the positions of obstacles with roadside infrastructure or other vehicles. However, full deployment of such communication systems still require considerable time for full implementation because of the extra equipment needed or replacements for existing vehicles or infrastructures. During this transition period, autonomous self-driving vehicles and vehicles driven by people will be sharing roads. Under such conditions, autonomous vehicles must be able to recognize traffic information, such as road signs and traffic lights, and respond accordingly.

Camera sensors are typically employed in object recognition tasks. For instance, several traffic light detection and recognition methods [45–49] and traffic sign recognition methods [50, 51] have been presented. Traffic light recognition systems for autonomous vehicles must be sufficiently fast and accurate. Generally, cameras installed in vehicles capture a wide variety of objects unrelated to the task of traffic light recognition, such as billboards and roadside trees; these objects are considered “noise” for traffic light

## 4.2. Proposed scheme and assumptions

---

recognition and can cause a significant reduction in the detection accuracy. High-resolution cameras can be used to improve recognition accuracy; however, they require longer processing times, which is not ideal for real-time driving. Most previous works contribute to the problem of traffic light state recognition using images only. Hence, their recognition performance would be decreased caused by noisy objects that unrelated to traffic light state recognition mentioned above. The negative effect from such unrelated objects will get worse in urban areas, whereas high-level autonomous driving systems should consider driving in such regions. Furthermore, state recognizers should be robust against the variety of visual appearance, including perspective and illuminance variation since these changes frequently occur in images captured by vehicle-installed cameras.

This chapter presents a review of a scheme to recognize traffic light states from images to demonstrate practical verification of GPU applicability to perception tasks. The scheme uses a region of interest (ROI) to extract only the image sections containing traffic lights so that robustness against unrelated noisy objects is maintained. To achieve this, localization results from the 3D map are used to extract the traffic light positions in 3D space. Using an intrinsically and extrinsically calibrated camera, the positions of these traffic lights are projected onto the image space, the surrounding areas in the image are extracted, and the data is fed to custom classifiers to obtain the traffic light status. GPUs are applied to one of the classifiers using deep-learning techniques which typically have a large capacity for perspective and illuminance variation. The results of quantitative evaluations indicate that the classifier using the deep-learning techniques achieve higher accuracies and are more reliable under preferable conditions compared with the classifier using traditional morphological processing.

## 4.2 Proposed scheme and assumptions

The proposed traffic light color state recognition method can be divided into the following subtasks:

- I) Extraction of regions from camera images that include traffic lights,

and

- II) Color state recognition using the extracted regions.

To clarify this problem, an overview of 3D maps and an ROI extraction technique are first presented. Then, the morphological and deep-learning traffic light state classification methods are discussed.

### 4.2.1 3D High Definition Maps

The proposed scheme assumes that the two 3D maps described below are given in advance. The first one is an accurate *3D point-cloud map*, which is an aggregation of measured points with 3D coordinate values that represent the shape information around the driving environment. Combining this map with the position estimation method described in the following section, the ego-vehicle's position on the map coordinate system can be obtained. The second one constitutes a *3D feature map* that is similar to the HERE HD Live Map [52]; it comprises detailed information (i.e., *high definition map features*) regarding traffic-related objects, including 3D position and orientation of traffic lights, traffic signs, and utility poles, as well as information for lane, road paint, and cross walk area. By putting them together the proposed scheme acquires ever-changing geometrical relationships between traffic objects to be recognized and the ego vehicle.

The 3D feature map used in this study mainly contains the following information for individual traffic lights:

- IDs for each of the bulbs comprising the traffic light,
- Horizontal and vertical angles in the direction that the traffic light bulb is facing,
- ID of the pole where the traffic light bulb is installed,
- Class of the traffic light bulb, i.e., pedestrian or traffic, and
- ID of the lane nearest to the traffic light.



Since the 3D feature map contains independent road features (i.e., points in space, vectors defining direction, traffic light bulbs, etc.), knowing that this feature map was built with a point cloud allows formation of complete instances of traffic lights and relating them to the vehicle driving path through comparisons with the lights in the direction facing away from the ego vehicle.

### 4.2.2 ROI Extraction

The ROI extraction is achieved with a camera, a LiDAR sensor, a localization method, and a map. A LiDAR is a sensor that emits ultraviolet, radiant, or infrared laser beams similar to radio waves used in conventional radar sensors. Similarly, it can be used to measure the distance between the sensor and other objects by analyzing the flight time of the emitted rays. Compared to radio waves, laser wavelengths are shorter by an order of magnitude, which enables a LiDAR to measure smaller objects and acquire detailed shape information. A 360° LiDAR sensor was employed to estimate the sensor position accurately in a given 3D point-cloud map by comparing the measured shape information to the map. Once the location was known on the point-cloud map, the corresponding 3D feature map was used to obtain the positions of the traffic lights. The ROI extraction process is as follows:

- Step 1) estimate the LiDAR position on a 3D map,
- Step 2) estimate the camera position on the 3D map,
- Step 3) project the 3D traffic light position coordinates onto a camera image,  
and
- Step 4) extract the ROI according to the projected traffic light position.

To estimate the LiDAR position on a 3D map (Step 1), the shape of the surrounding environment is measured first. The LiDAR position on the 3D map is acquired by comparing this shape to a 3D point-cloud map using normal distribution transform (NDT) [53, 54]. During experimentation, it was observed that NDT could estimate the LiDAR position accurately. The

range of localization errors is of the order of centimeters; therefore, it can be assumed that the localization precision is acceptable on 3D maps.

The proposed method also requires the 3D positional relationship between the LiDAR and camera (also referred to as “extrinsic parameters”). We employ the method proposed in [55] to acquire this relationship. This work obtains these extrinsic parameters from multiple LiDAR sensors using multiple planes. According to the reported results, the calibration results contain certain errors but these are almost negligible.

Bearing in mind that both localization and calibration contain errors, the proposed method is designed to be resilient to errors to a certain degree. However, as discussed in Section 4.4, excessive calibration errors will reduce the recognition accuracy because the ROI extraction would fail to capture the traffic lights in the ROIs.

The positional relationships between the LiDAR sensor and camera are calculated in advance and used to estimate the camera position on the map (Step 2). This relationship has six degrees of freedom that represent translations and rotations in 3D space. These extrinsic parameters remain constant as long as the sensors are fixed to the vehicle body. The camera position on the 3D map is calculated using the location obtained by NDT (Step 1), and the relative position between the LiDAR and camera sensor is given as follows:

$$p_{cam} = R(\alpha, \beta, \gamma) \cdot p_{LID} + T(x, y, z) \quad (4.1)$$

where:

- $p_{cam}$  denotes the camera position in the 3D map coordinate system,
- $p_{LID}$  denotes the position of the LiDAR sensor in the 3D map coordinate system,
- $(x, y, z)$  denotes the relative translations,
- $(\alpha, \beta, \gamma)$  denotes the relative rotation angles about each axis,
- $T(x, y, z)$  is a translation matrix, and
- $R(\alpha, \beta, \gamma)$  is a rotation matrix.

To project the 3D traffic light position coordinates on a camera image (Step 3), traffic light coordinates in the 3D camera coordinate system

## 4.2. Proposed scheme and assumptions

---

$(s_x, s_y, s_z)$  are obtained via transformation into the 3D map coordinate system. These coordinates are projected onto the image plane coordinates  $(u, v)$  using the camera's focal length  $(f_x, f_y)$  and center of the image  $(c_x, c_y)$  as follows.

$$u = s_x \frac{f_x}{s_z} + c_x, v = s_y \frac{f_y}{s_z} + c_y \quad (4.2)$$

The ROIs are extracted from the images (Step 4) according to the coordinates calculated in Step 3. Note that the ROI extraction process tolerates a certain level of numerical errors that occur in Step 1–Step 3 and the extrinsic calibration error mentioned above. Fig. 4.1 shows the specifications of the ROI with some margins. In the proposed scheme, the radius of each traffic light blob in the real world is assumed as 30 cm. By exploiting the 3D pose of the traffic lights and a vehicle-mounted camera on the 3D map, the radius of the traffic light projected on the image plane can be estimated ( $r$  in Figure 4.1). To tolerate the numerical errors, a margin of  $1.5r$  from the projected edge of the traffic light was adopted. An overview of the ROI extraction technique used in the proposed scheme is shown in Fig. 4.2.

### 4.2.3 Morphological Processing

Morphological processing manipulates and analyzes the brightness values of pixels in the ROI, as explained in Section 4.2.2.

First, images are converted from the RGB to HSV color space. The HSV color space is more closely related to human chromatic sensation than the RGB space; thus, the conversion allows easier determination of the color value thresholds. A mask image is then generated using the H, S, and V threshold conditions set for each traffic light color to extract the regions that include a colored (illuminated) light. The generated candidate regions may contain areas that do not correspond to a traffic light because some pixel blocks that satisfy the threshold conditions but are not actually traffic lights could be from other objects in the images. We assume that the traffic lights projected onto the image plane are round; therefore, an additional threshold condition based on the degree of circularity is applied to each candidate region. The

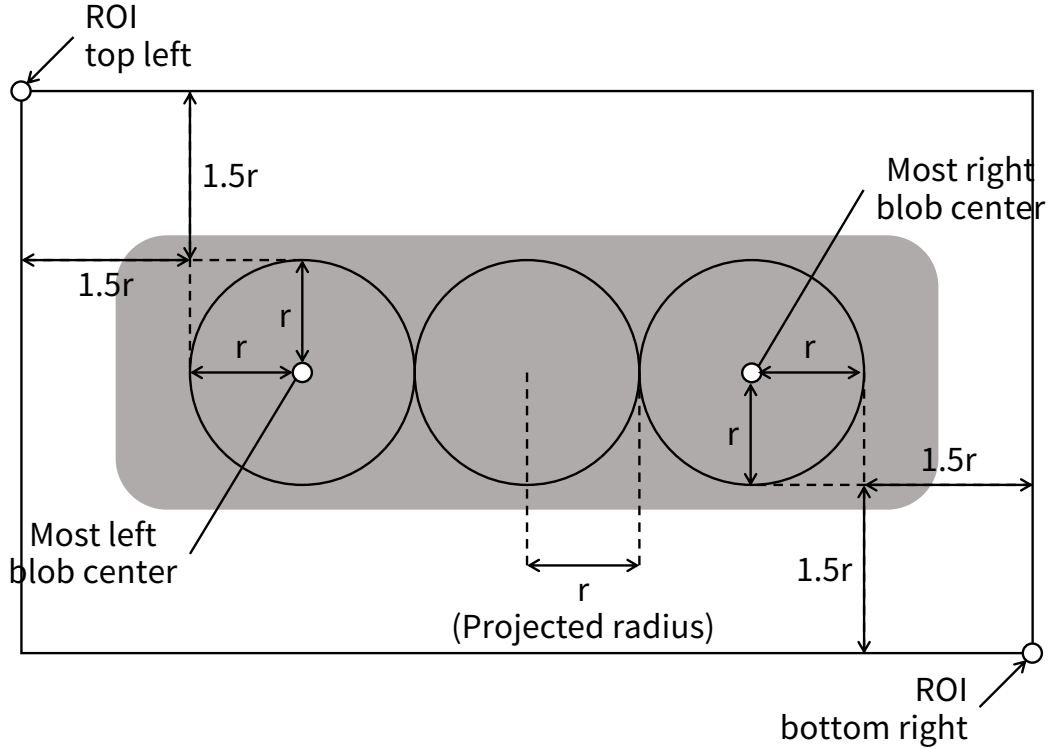


Figure 4.1: Proposed ROI extraction definition.

degree of circularity  $R$  is expressed as follows:

$$R = \frac{4\pi \times S}{L^2} \quad (4.3)$$

where  $S$  and  $L$  denote the area and perimeter of the region, respectively. The degree of circularity represents the extent to which a region is circular (values closer to 1 indicate a region more similar to a circle). If several candidate regions remain after applying the degree of circularity threshold, the region with the highest degree of circularity is selected and a mask is applied to filter areas outside the region. Table 4.1 shows the threshold values that were experimentally determined for the morphological processing method.

This mask image is overlaid on the input ROI to obtain the pixel values assumed to represent the traffic light. This recognition process infers the color state of a target traffic light by searching for the most dominant color in the range of pixel values. Fig. 4.3 shows the processing workflow up to this point.

## 4.2. Proposed scheme and assumptions

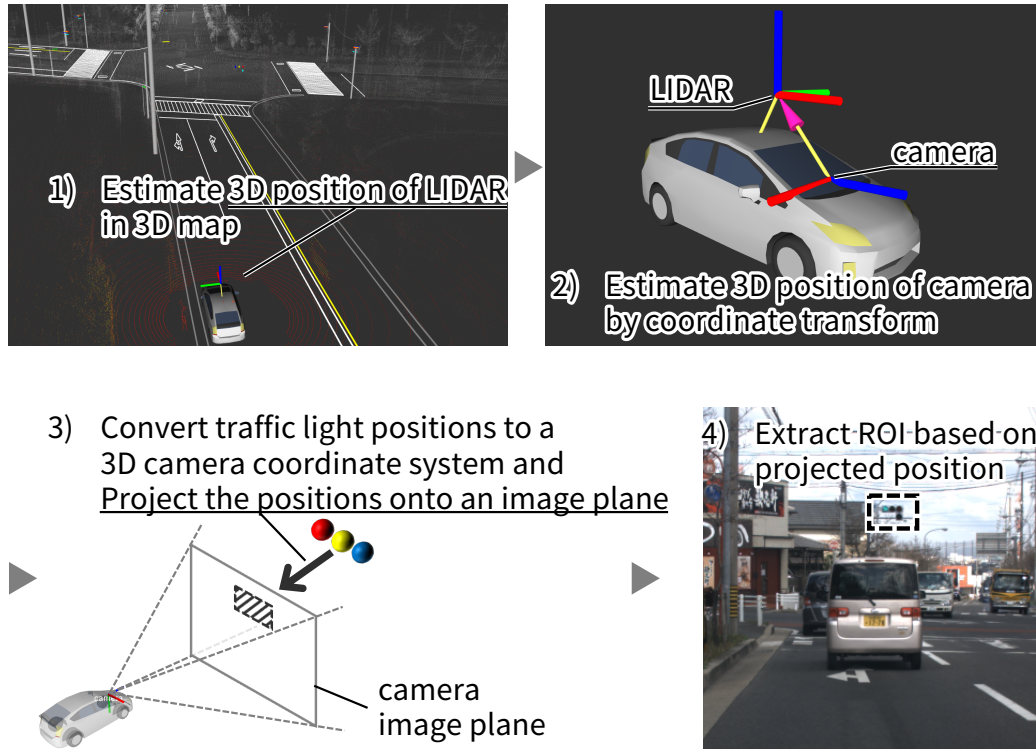


Figure 4.2: ROI extraction overview.

### 4.2.4 Deep-Learning-Based Detector

Deep CNNs may be applied to obtain the location and class of the traffic signals. An SSD [28] was used in this work. The SSD was originally developed as an object detection method using CNN, and one of its remarkable features is the fast inference time. Moreover, the SSD detection accuracy is comparable to those of other state-of-the-art object detection algorithms (e.g., Faster-RCNN [43]) and works effectively with lower resolution images because it exploits hierarchical feature maps. While the shapes of the traffic lights in the extracted ROIs (Section 4.2.2) are almost similar, the SSD can distinguish each traffic light color state as a different object because the color element weights are increased when sufficient samples are provided during the training phase. Hence, this object detection algorithm was applied to traffic light color state recognition.

As discussed in Section 4.1, the traffic light recognition techniques in autonomous driving require high processing speeds to recognize the traffic

CHAPTER 4. TRAFFIC LIGHT RECOGNITION USING HIGH  
DEFINITION MAP FEATURES

Table 4.1: Threshold values used in morphological processing.

Note that “H” is represented in the range of 0 to 360 cyclically, while “S”, “V”, and “R” are in the linear range of 0 to 1.

Light color	H		S	V	R
	lower	upper			
Red	340	50			
Yellow	50	70	0.37	0.55	0.75
Green	80	190			

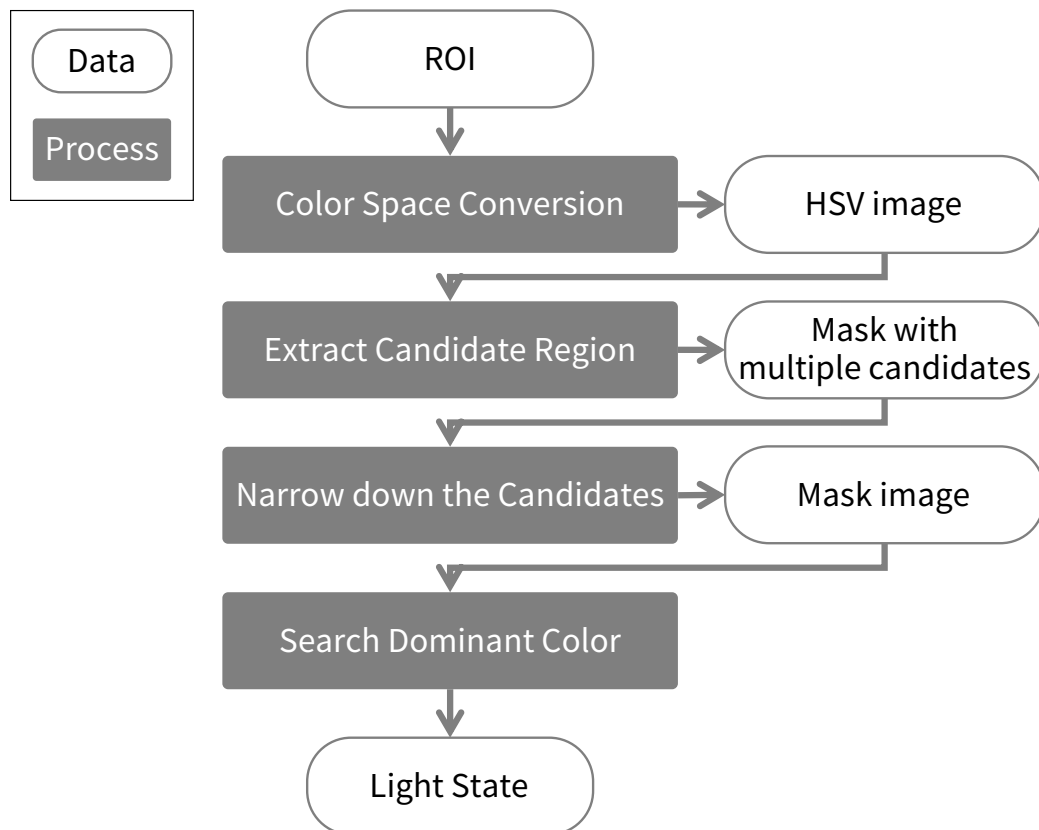


Figure 4.3: Flow diagram for morphological recognition.

lights in front of a vehicle traveling at typical speeds, e.g., 60 km/h in Japan. High recognition accuracy is also required because the results significantly influence vehicle control (e.g., deceleration when approaching intersections).

## 4.2. Proposed scheme and assumptions

---

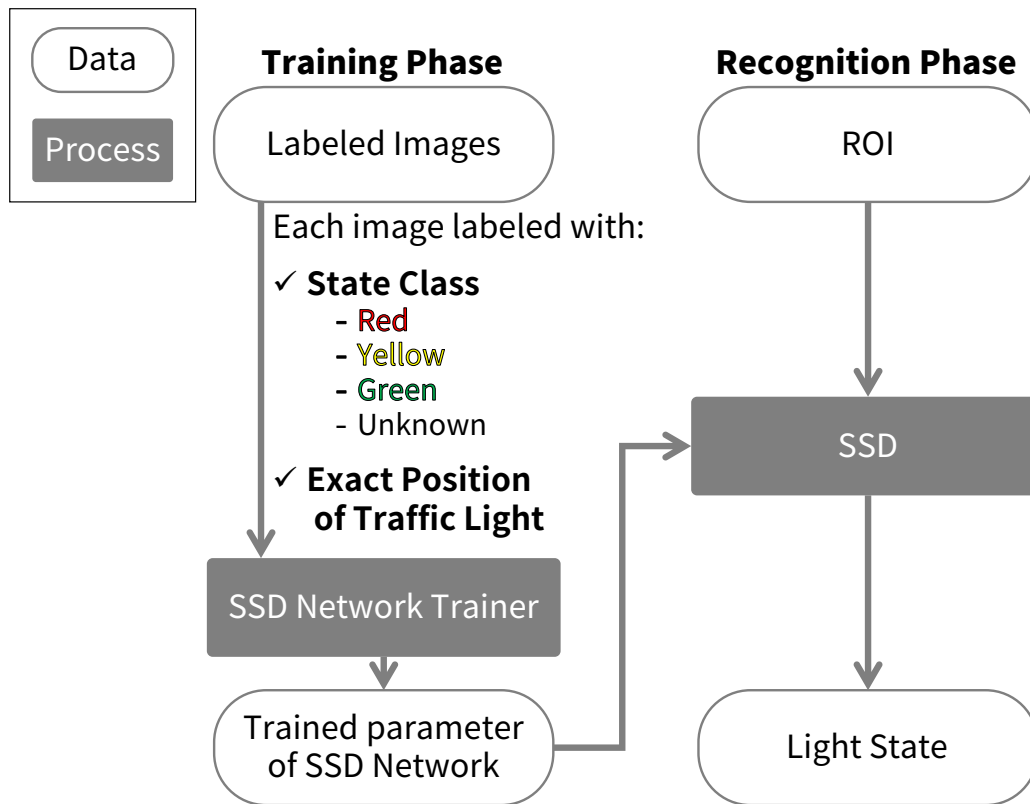


Figure 4.4: Flow of recognition with the SSD approach.

The size of the ROI image (Section 4.2.2) is reduced when the traffic lights are projected onto the image plane from greater distances. It is therefore preferable for a traffic light recognition algorithm to work with low-resolution inputs to plan vehicle behaviors in advance. Hence, the SSD algorithm can be considered suitable for this purpose. The recognition flow with the SSD algorithm is shown in Fig. 4.4. Note that the SSD outputs bounding boxes (i.e., locations of objects) and classes since it is an object detection method, whereas the objective of this study is to only acquire the classes (states) of traffic lights inside a given ROI. Therefore, all bounding boxes from the SSD results are ignored and the class with the highest detection score is adopted as the recognized state for the ROI.

## 4.3 Implementation

This section presents the detailed implementation for the traffic light color recognition described in Section 4.2.

### 4.3.1 Autoware Implementation

All functions described in Section 4.2 were implemented using Autoware [56, 57], which is an open-source research and development platform for autonomous driving based on the robot operating system (ROS) [58]. Autoware modularizes the functions required for autonomous driving as individual processes, and ROS provides efficient interprocess communication, which allows users to send and receive user-defined data structures flexibly.

The proposed traffic light recognition module employed the modules integrated with Autoware for localization and 3D Maps to achieve the expected functionality. Fig. 4.5 shows an overview of the connections between the data acquired by the Autoware modules and the proposed method’s functionality.

As shown on the right side of Fig. 4.5, the proposed method is divided into two parts (i.e., “Feature Projection” and “Light State Recognition”), which were implemented as individual ROS nodes. The “Feature Projection” process performs ROI extraction using a 3D map and the localization result (Section 4.2.2); this process receives data from Autoware, such as camera information (image size and focal length), traffic light coordinates in a 3D map, positional relationship between the vehicle-mounted LiDAR and camera sensor, and estimated LiDAR position on the 3D map. Then, it outputs the ROI information, which indicates the positions of the traffic lights in the given image.

The “Light State Recognition” process corresponds to morphological processing (Section 4.2.3) and deep learning (Section 4.2.4). These nodes receive the ROI information obtained by the feature projection process and images captured by the camera. The target region is extracted from the camera image according to the ROI information, followed by the recognition process. Note that the formats of the input and output data are uniform for reusability among all recognition methods that comply with the defined



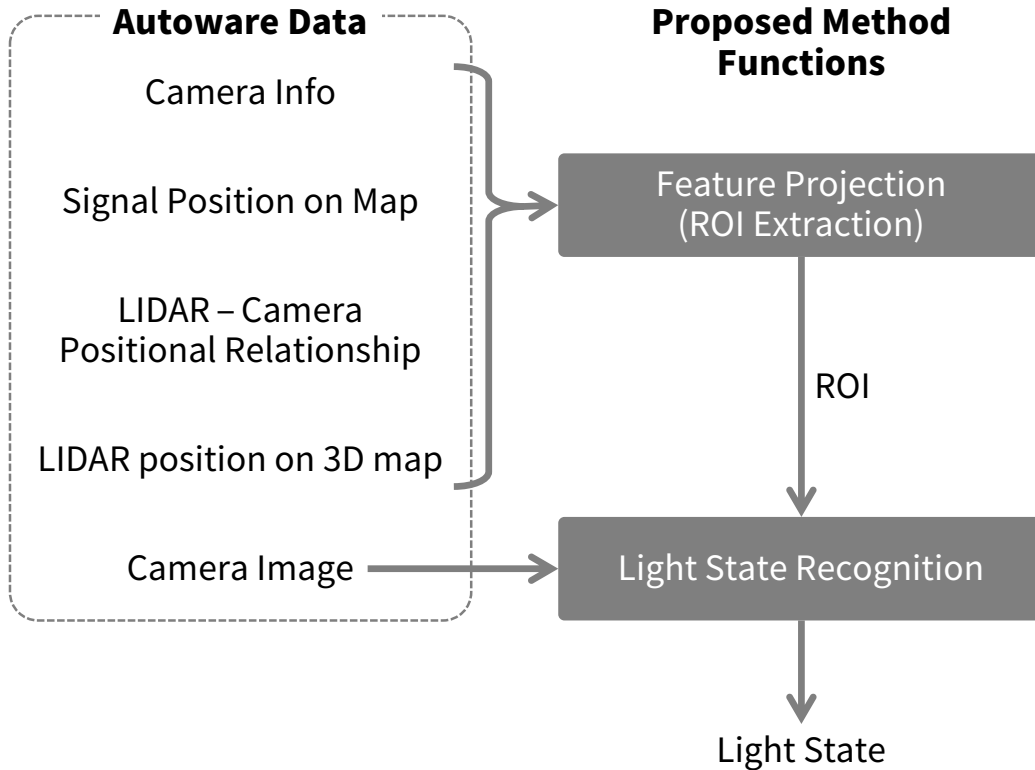


Figure 4.5: Overview of data connections between Autoware and the proposed method.

messages.

### 4.3.2 Color State Training and Recognition by SSD

The Caffe [59]-based implementation<sup>1</sup> of SSD (Section 4.2.4) was used as the basis for this work. As shown in the “Training Phase” (Fig. 4.4), the traffic light color states and traffic light positions in each image were used as the training data when the SSD learned the network parameters. Noise, including other vehicles in the front of the vehicle and/or roadside trees due to vibration while driving and occlusions may be present in these ROIs acquired from real autonomous driving environments. Therefore, to create a training dataset, images were collected using a vehicle-mounted camera during on-road driving experiments and labeled with the color state and traffic light

<sup>1</sup><https://github.com/weiliu89/caffe/tree/ssd>

CHAPTER 4. TRAFFIC LIGHT RECOGNITION USING HIGH DEFINITION MAP FEATURES

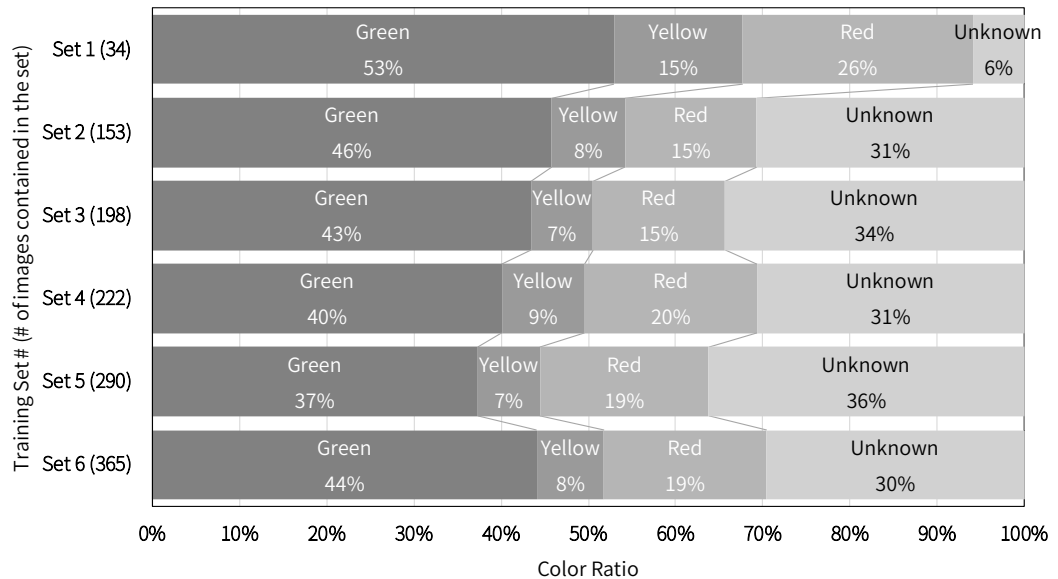


Figure 4.6: Breakdown of the training dataset.

position information. Some training data examples are shown in Table 4.2. During the SSD training phase, the exact bounding box coordinates enclosing the traffic lights in the ROI were fed along with the color states. This is similar to general training schemes for object detection methods. The aim here was for the SSD to learn the features of the traffic lights without the margin areas in the extracted ROI. In other words, the aim was to degrade the sensitivity to background features. Fig. 4.6 shows the training datasets and a breakdown of the color states considered in this work.

Although a previous study [28] reported that the SSD algorithm can achieve a higher detection rate with low-resolution input compared with state-of-the-art detection algorithms, including Fast R-CNN [60] and YOLO [27], the lowest image resolution considered in the study was  $300 \times 300$  pixels. However, a traffic light recognition system for autonomous driving must frequently handle smaller resolution images (Table 4.2) for reasons described in Section 4.2.4. Evaluations of the recognition accuracy transition are discussed using multiple input resolutions in Section 4.4.

### 4.3. Implementation

Table 4.2: Training data examples for traffic light recognition.

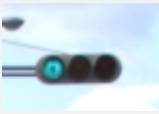

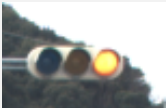

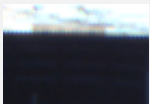
Image	Size $\begin{bmatrix} \text{width} \\ \text{height} \end{bmatrix}$	State	Exact Light Position		Remarks (Not used in training)
			$\begin{bmatrix} x \\ y \\ \text{width} \\ \text{height} \end{bmatrix}$		
	$\begin{bmatrix} 74 \\ 50 \end{bmatrix}$	Green	$\begin{bmatrix} 15 \\ 23 \\ 57 \\ 38 \end{bmatrix}$		–
	$\begin{bmatrix} 52 \\ 35 \end{bmatrix}$	Yellow	$\begin{bmatrix} 5 \\ 12 \\ 36 \\ 24 \end{bmatrix}$		–
	$\begin{bmatrix} 76 \\ 50 \end{bmatrix}$	Red	$\begin{bmatrix} 11 \\ 19 \\ 57 \\ 36 \end{bmatrix}$		–
	$\begin{bmatrix} 109 \\ 70 \end{bmatrix}$	Unknown	$\begin{bmatrix} 31 \\ 23 \\ 93 \\ 46 \end{bmatrix}$		Turned off owing to camera frame rate and signal blinking frequency
	$\begin{bmatrix} 68 \\ 46 \end{bmatrix}$	Unknown	$\begin{bmatrix} 0 \\ 0 \\ 68 \\ 46 \end{bmatrix}$		Occluded by vehicle in front

Table 4.3: Interframe filter for morphological processing.

Each digit of the “GYR result of current frame” is 1 if there is a region that fulfills the conditions for the corresponding color (green, yellow, and red) in a current ROI.

Previous result	GYR result of current frame							
	000	001	010	011	100	101	110	111
Green	Green	UNK	Yellow	Yellow	Green	Green	Yellow	UNK
Yellow	Yellow	Red	Yellow	Red	UNK	UNK	Yellow	UNK
Red	Red	Red	UNK	Red	Green	Red	Green	UNK
UNK	UNK	Red	Yellow	Red	Green	Red	Yellow	UNK

\* UNK: Unknown

### 4.3.3 Interframe Filter

As traffic lights are designed for human visual assessments, it can be assumed that the color state does not change faster than a camera’s frame rate. Moreover, the general traffic lights in Japan change their color states in the order of green, yellow, and red. By considering these assumptions, the original filters [61] were modified to reduce false recognitions in a few frames; the filters were designed to consider the order of color state changes. The final output recognition results do not change until a fixed number of the same recognition results have been acquired with the filter. Note that different filters were created for the morphological processing and deep-learning recognizers (Sections 4.2.3 and 4.2.4, respectively) because each method demonstrated different recognition tendencies.

Table 4.3 details the interframe filter for the morphological recognizer, with which all the traffic light colors (green, yellow, and red) are estimated independently. In this process, a threshold-based assessment is primarily used to estimate the color pixel blocks that correspond to a traffic light. As discussed in Section 4.2.3, some blocks may fulfill the HSV and degree of circularity threshold conditions and eventually be considered final candidates. In Table 4.3, combinations of the estimates of each of the light colors in a

#### 4.4. Evaluation of the proposed scheme using practical data

---

Table 4.4: Interframe filter for machine learning recognition method.

Previous result	Result of current frame			
	Green	Yellow	Red	Unknown
Green	Green	Yellow	Yellow	Green
Yellow	Unknown	Yellow	Red	Yellow
Red	Green	Red	Red	Red
Unknown	Green	Yellow	Red	Unknown

current ROI are represented by the “GYR result of current frame”. For example, “110” indicates that the input ROI contains regions that satisfy the conditions for luminous green and yellow lights. The interframe filter returns the current color state by comparing the previous recognition result to that of the current frame.

Table 4.4 shows the interframe filter for the SSD recognizer, which is an end-to-end process that takes an image as input and outputs the recognition result; thus, its recognition results are limited to only four patterns: green, yellow, red, and unknown. In addition, the SSD recognizer can improve the recognition accuracy for an individual frame via learning, whereas the morphological recognizer cannot. Thus, compared to the morphological recognizer’s interframe filter, the SSD recognizer’s interframe filter better upholds the recognized suggestion for the current frame as the final result (e.g., the SSD recognizer’s suggestion for the current frame can be the final result more straightforwardly as compared with the morphological recognizer’s suggestion).

## 4.4 Evaluation of the proposed scheme using practical data

To quantify the performance of the proposed traffic light recognition scheme, the effects of the following factors were evaluated: 1. ROI extraction,

Table 4.5: Evaluation dataset.

	Daytime	Sunset
Driving duration	596 s	328 s
Image resolution	1368 (w) × 1096 (h)	1368 (w) × 1096 (h)
# of frames	8946	4929
# of extracted frames	3347	1505
# of target signals	8	5
# of state changes	6	4

2. Distance from the recognition target, 3. GPU usage, and 4. Number of images in the training dataset for the SSD recognizer.

#### 4.4.1 Experimental Setup

An Intel Core i7-6700K operating at 4.0GHz with four cores was used as the host processor. The GPU used for program acceleration was an NVIDIA GeForce GTX 980 Ti (CUDA version 8.0). As the evaluation dataset, captured images from a vehicle-mounted camera during driving on public roads were used. Image collection was performed on several roads in the morning and at sunset. The evaluation dataset was then created by extracting frames in which target traffic lights were present. An overview of the evaluation dataset is shown in Table 4.5. A breakdown of the color states in each evaluation dataset is shown in Fig. 4.7.

#### 4.4.2 Effect of ROI Extraction on Recognition Accuracy

Fig. 4.8 shows the recognition accuracy for the daytime and sunset datasets obtained by the two recognition methods. The term “Accuracy” represents the number of recognition results that agree with the ground truth states of the evaluation dataset (expressed as a percentage). ROI extraction improves the accuracy for nearly all datasets and recognition method combinations. Therefore, it is concluded that confining the processing region by extracting

#### 4.4. Evaluation of the proposed scheme using practical data

---

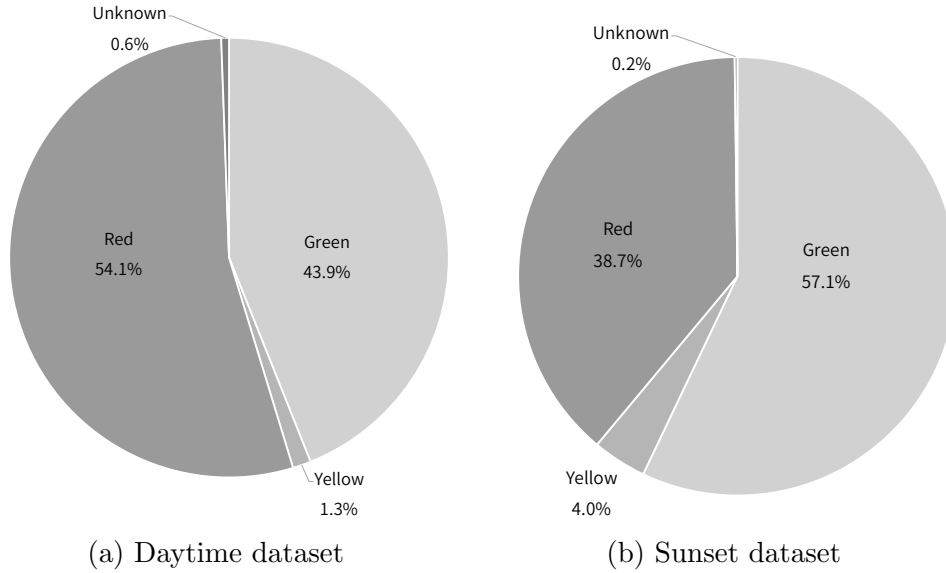


Figure 4.7: Proportions of the ground truth images in each evaluation dataset.

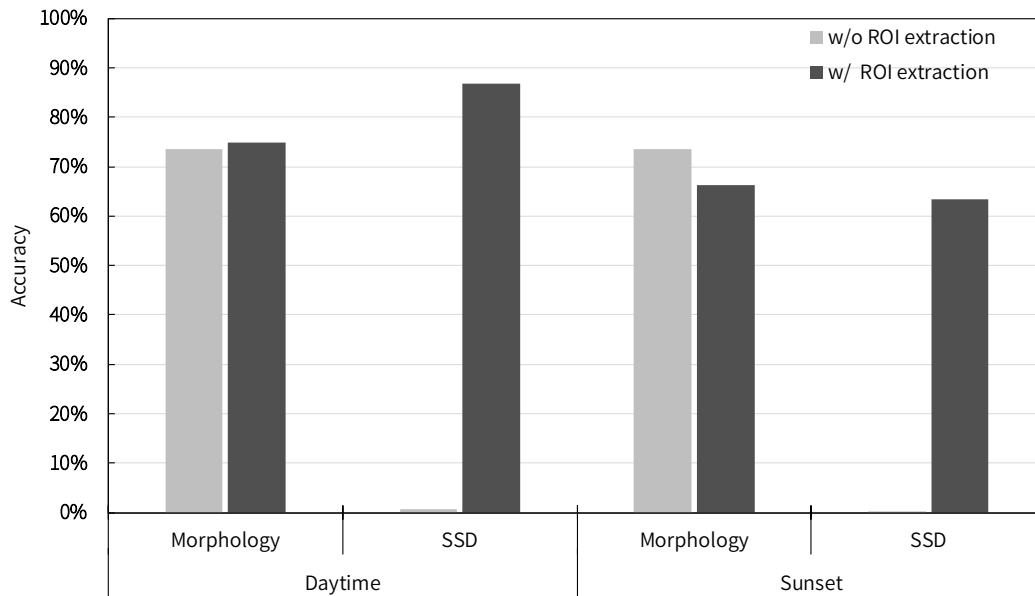


Figure 4.8: Effects of ROI extraction on recognition accuracy.

an ROI is beneficial for traffic light recognition. The accuracy of the SSD recognizer with the evaluation dataset was up to 86.9%, which is a significant improvement. Note that Set 6 (Fig. 4.6) was used for the following SSD recognizer evaluations unless noted otherwise.

Fig. 4.9 shows the recognition precision (bars) and recall (lines). Here, “Precision” is the proportion of the number of correct results with respect to all recognizer outputs, and “Recall” is the proportion of the number of correct recognitions with respect to the total number of images whose ground truths correspond to the states in the evaluation dataset. Tables 4.6 and 4.7 show the detailed information for the recognition results expressed as confusion matrices.

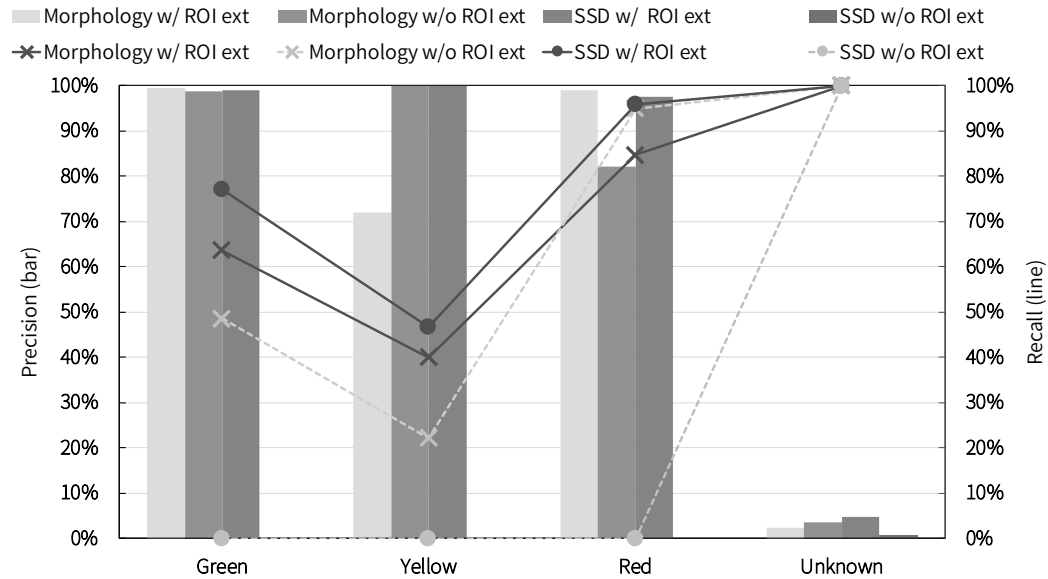
As shown in Tables 4.6(d) and 4.7(d), the SSD recognizer without ROI extraction outputs “Unknown” for all inputs. This is assumed to be caused by the clipped-out images used to train the SSD network (Table 4.2). Note that the SSD recognizer without ROI extraction is omitted from the subsequent discussion because all of its outputs were “Unknown”, which makes it impossible to discuss its recognition tendencies.

Fig. 4.9(a) shows that all recognizers demonstrated similar tendencies, i.e., the recall was lowest for the yellow state and increased in ascending order for the green and red states. The evaluation dataset inherently contains fewer yellow state images than the other states; thus, a moderate false recognition may have caused a significant reduction of the recall proportion. For the SSD recognizer, the lower recall for the yellow state was presumably due to the fact that the training dataset included fewer yellow state images, as shown in Fig. 4.6. The morphological recognizer without ROI extraction produced a lower precision for the red state than that with ROI extraction for the daytime dataset. In addition, by comparing the “Red” columns in Tables 4.6(a) and 4.6(b), the morphological recognizer without ROI extraction was observed to be predisposed to output red for any input. This tendency seems to indicate that processing without ROI extraction caused false recognition because the recognizer considered non-traffic light regions (e.g., the tail lights of other vehicles) as red traffic lights.

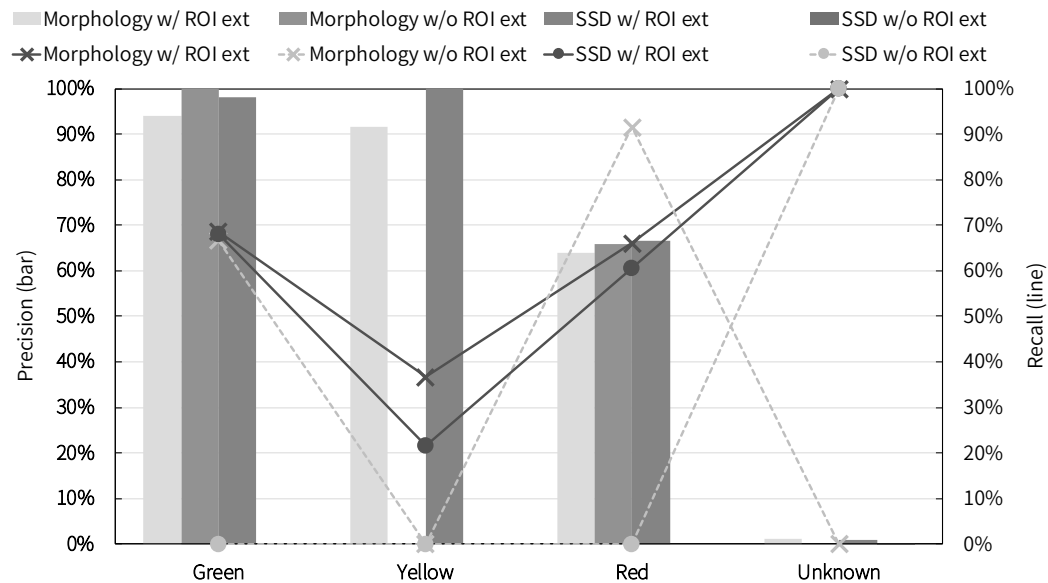
In contrast, as shown in Fig. 4.9(b), the recall values of all the recognizers were reduced for the sunset evaluation dataset. This may have been caused



#### 4.4. Evaluation of the proposed scheme using practical data



(a) Results for the daytime dataset



(b) Results for the sunset dataset

Figure 4.9: Recognition precision and recall by color for each dataset.

CHAPTER 4. TRAFFIC LIGHT RECOGNITION USING HIGH  
DEFINITION MAP FEATURES

---

Table 4.6: Confusion matrices for each recognition method (daytime dataset).

(a) Morphology w/ ROI extraction

		Prediction				Recall
		Green	Yellow	Red	Unknown	
Ground Truth	Green	936	0	17	518	63.6%
	Yellow	6	18	0	21	40.0%
	Red	0	7	1536	269	84.8%
	Unknown	0	0	0	19	100.0%
Precision		99.4%	72.0%	98.9%	2.3%	

(b) Morphology w/o ROI extraction

		Prediction				Recall
		Green	Yellow	Red	Unknown	
Ground Truth	Green	715	0	363	393	48.6%
	Yellow	9	10	13	13	22.2%
	Red	0	0	1721	91	95.0%
	Unknown	0	0	0	19	100.0%
Precision		98.8%	100.0%	82.1%	3.7%	

(c) SSD w/ ROI extraction

		Prediction				Recall
		Green	Yellow	Red	Unknown	
Ground Truth	Green	1132	0	44	295	77.0%
	Yellow	11	21	0	13	46.7%
	Red	0	0	1735	77	95.8%
	Unknown	0	0	0	19	100.0%
Precision		99.0%	100.0%	97.5%	4.7%	

(d) SSD w/o ROI extraction

		Prediction				Recall
		Green	Yellow	Red	Unknown	
Ground Truth	Green	0	0	0	1471	0.0%
	Yellow	0	0	0	45	0.0%
	Red	0	0	0	1812	0.0%
	Unknown	0	0	0	19	100.0%
Precision		0.0%	0.0%	0.0%	0.6%	

#### 4.4. Evaluation of the proposed scheme using practical data

---

Table 4.7: Confusion matrices for each recognition method (sunset dataset).

(a) Morphology w/ ROI extraction

		Prediction				Recall
		Green	Yellow	Red	Unknown	
Ground Truth	Green	589	0	217	53	68.6%
	Yellow	38	22	0	0	36.7%
	Red	0	2	385	196	66.0%
	Unknown	0	0	0	3	100.0%
Precision		93.9%	91.7%	64.0%	1.2%	

(b) Morphology w/o ROI extraction

		Prediction				Recall
		Green	Yellow	Red	Unknown	
Ground Truth	Green	573	0	215	71	66.7%
	Yellow	0	0	60	0	0.0%
	Red	0	0	534	49	91.6%
	Unknown	0	0	3	0	0.0%
Precision		100.0%	0.0%	65.8%	0.0%	

(c) SSD w/ ROI extraction

		Prediction				Recall
		Green	Yellow	Red	Unknown	
Ground Truth	Green	585	0	141	133	68.1%
	Yellow	11	13	36	0	21.7%
	Red	0	0	353	230	60.5%
	Unknown	0	0	0	3	100.0%
Precision		98.2%	100.0%	66.6%	0.8%	

(d) SSD w/o ROI extraction

		Prediction				Recall
		Green	Yellow	Red	Unknown	
Ground Truth	Green	0	0	0	859	0.0%
	Yellow	0	0	0	60	0.0%
	Red	0	0	0	583	0.0%
	Unknown	0	0	0	3	100.0%
Precision		0.0%	0.0%	0.0%	0.2%	

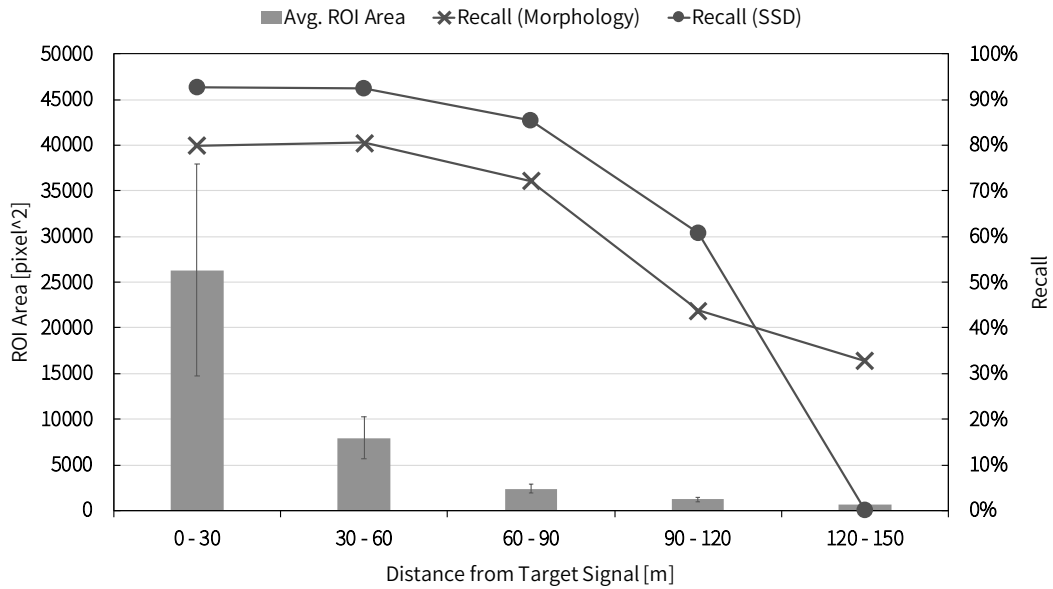
by pixel value saturation resulting from strong solar backlight. Even if the sunlight was not directly incident on the camera, significant HSV changes caused by automatic white-balance corrections triggered by an overly bright backlight could be another reason for the reduced recall. Regardless of whether ROI extraction was employed, the morphological recognizers were predisposed to incorrectly recognize the green state as red state, as suggested by a comparison of the “Green” rows in Tables 4.7(a) and 4.7(b). This implies that the morphological recognizers have less flexibility relative to environmental changes, such as strong backlight. On the other hand, Table 4.7(c) shows that the SSD recognizer achieves higher precisions for nearly all the states than the morphological recognizer for the same input images. It is conceivable that if appropriate data are included in the training dataset, then the SSD recognizer can identify traffic lights using other information, such as shapes and the order of colors, when the HSV values change slightly owing to the backlight. Thus, collecting a feasible training dataset is important for the SSD recognizer.

### 4.4.3 Effects of Distance from Target on Recognition Recall

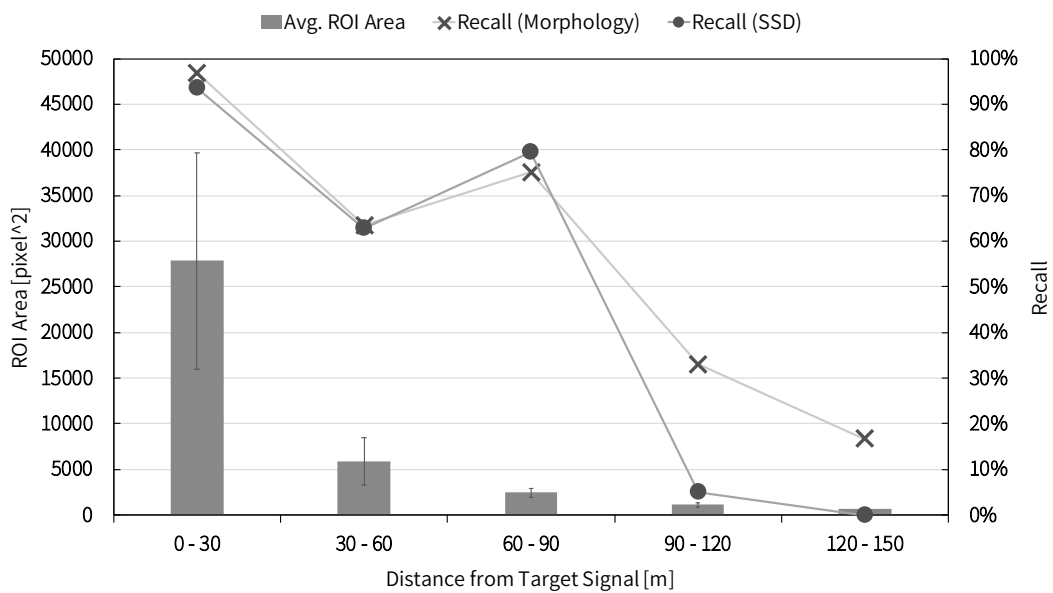
In Fig. 4.10, the horizontal axis is the distance from the target traffic light to be recognized, the left vertical axis is the area (square pixels) of the extracted ROI, and the right vertical axis is the recognition recall. Here, both recognizers (morphological and SSD) used the ROI images as inputs.

As shown in Fig. 4.10(a), the ROI area and recognition recall for both recognizers increase with decreasing distances from the target traffic lights. For intervals less than 120 m from the target, both recognizers yielded similar curves and converged to their maximum recall values. In contrast, the morphological recognizer outperformed the SSD recognizer in terms of recall in the interval from 120 m to 150 m from the target. The mean area of the ROI in this interval for the daytime dataset was approximately 673.5 square pixels, which means that each ROI had approximately 25 pixels to a side. In this case, the target traffic lights were represented by only a few pixels, and the available features of the traffic lights were limited to color

#### 4.4. Evaluation of the proposed scheme using practical data



(a) Results for the daytime dataset



(b) Results for the sunset dataset

Figure 4.10: Recognition recall shifts relative to distances from the target traffic lights.

information only. The SSD recognizer seemed to learn the color information and features, such as traffic light edge information. The recall inversion in the interval from 120 m to 150 m from the target was presumably caused by the lack of edge information and by confining the determination criteria to only the color information.

In contrast to Fig. 4.10(a), the morphological recognizer often outperformed the SSD recognizer relative to recalls for the sunset dataset (Fig. 4.10(b)). A possible reason for this is the errors in traffic light projection to an image plane when extracting the ROIs. In the sunset evaluation dataset, many traffic lights were observed only after curves in the roads. If these traffic lights are projected onto an image plane while navigating the vehicle through curves, the traffic light was not completely captured in the ROI owing to LiDAR-camera calibration errors. In this case, the morphological recognizer could identify the traffic light states as long as lit lights were included in the ROIs, while the SSD recognizer was unable to obtain the same results because it uses additional information (other than color) to recognize the states. In the interval from 30 m to 60 m from the target, the recalls of both recognizers were reduced. This is presumably because of solar glare through the cloud cover that was captured by the camera in this distance interval. Moreover, some lit lights were not contained in the ROI owing to the ROI extraction error mentioned previously, which rendered it difficult for both the recognizers to identify the traffic lights. The proposed scheme only focuses on the software processes, which assume that input images are given. A strategy combined with hardware functions, including automatic camera exposures, may be a promising solution to improve the robustness of the proposed scheme under difficult-to-recognize conditions, such as in the sunset dataset used in this work.

#### 4.4.4 Effects of ROI Extraction on Recognition Speed

Fig. 4.11 shows the recognition time of each recognizer with and without ROI extraction. The average recognition time of the morphological recognizer without ROI extraction was approximately 84.9 ms (approximately 0.51 ms with ROI extraction). For the SSD recognizer, the average recognition time

#### 4.4. Evaluation of the proposed scheme using practical data

---

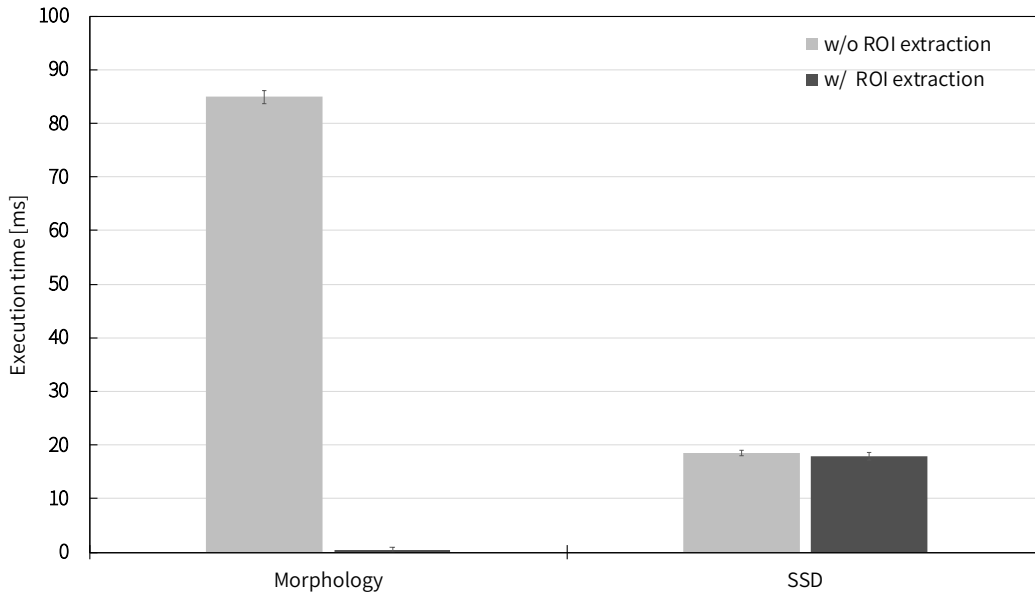


Figure 4.11: Effects of ROI extraction on recognition time.

was approximately 18.5 ms and 17.9 ms without and with ROI extraction, respectively. Note that the time required for ROI extracting and processing is not included in these recognition times. The morphological recognizer improved the recognition speed by approximately 166.5 times when using ROI extraction because it contains a process that depends on the input image resolution, e.g., color conversion from RGB to HSV and searching contours. In contrast, the SSD recognizer demonstrated similar recognition times regardless of whether ROI extraction was employed. This was owed to the fact that the SSD recognizer considers fixed-size images as inputs, which makes the processing time independent of the input image resolution because the input images are resized to fixed dimensions.

#### 4.4.5 Effects of GPU on SSD Recognition Speed

In Fig. 4.12, the execution time of the SSD recognizer with GPU acceleration is compared with those without a GPU. The execution time of the SSD recognizer with ROI extraction and GPU acceleration was approximately 17.9 ms (18.5 ms without ROI extraction). On the other hand, if GPU acceleration was not available, the execution times of the SSD recognizer for

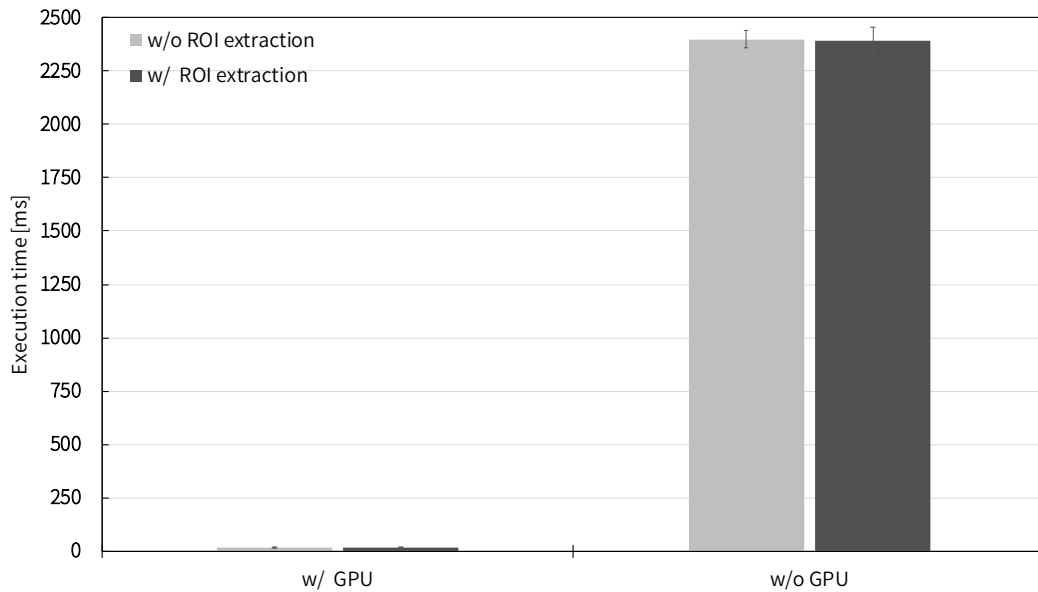


Figure 4.12: Effects of GPU on recognition time.

the two settings were approximately 2392.2 ms and 2397.9 ms. As noted in the previous section, the SSD execution time is not dependent on input image resolution because the images are resized to a fixed resolution. However, the experimental results indicate a 133.6 times poorer execution time in the absence of GPU acceleration. Therefore, GPU acceleration is necessary for applying SSD to autonomous driving. In addition, the acceleration rates vary significantly according to the type of GPU used [62]. Thus, appropriate GPUs must be selected for practical use in consideration of energy consumption and sufficient processing speed.

#### 4.4.6 Effects of the Number of Images in the Training Data on Recognition Accuracy

Variations in the recognition accuracies of the SSD recognizer relative to the number of images in the training dataset are shown in Fig. 4.13. The six sample points correspond to the recognition accuracies of the results obtained with the training dataset shown in Fig. 4.6. Briefly, a training dataset with more images results in greater accuracy. The accuracy improvement appears to converge up to Set 5 (Fig. 4.6), which contains 290 images. The training



#### 4.4. Evaluation of the proposed scheme using practical data

---

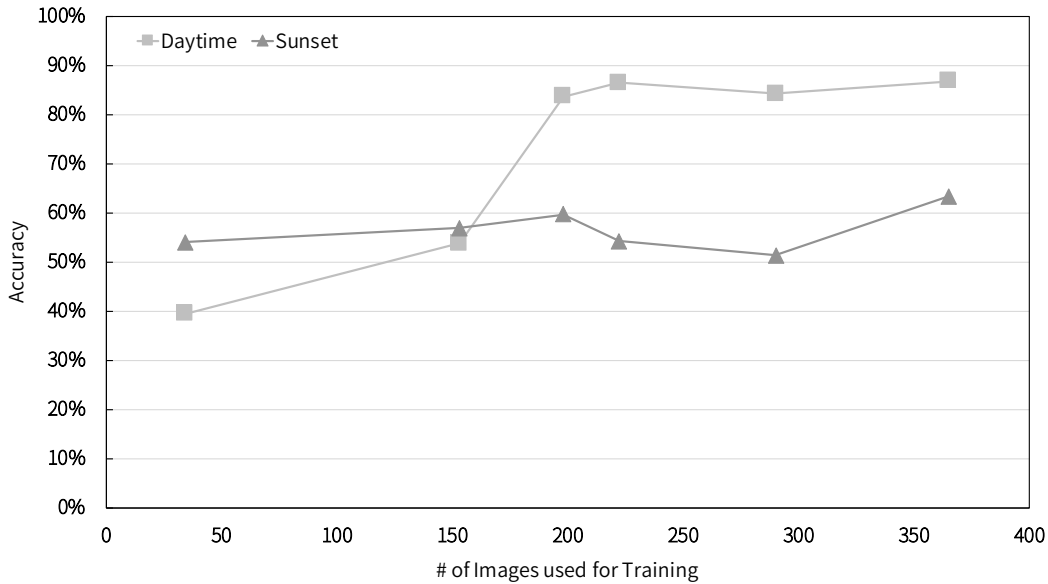


Figure 4.13: Recognition accuracy shifts relative to the number of images in the training dataset.

datasets for Sets 1 to 5 include images extracted from data recorded during daytime driving. In contrast, Set 6 contained both daytime and traffic lights images exposed to solar backlight. Recognition accuracy improvements were observed with both datasets when the training results obtained using Set 6 were applied to the SSD recognizer. Thus, the data quantity, diversity, and practicality relative to the recognition targets in the training datasets contribute considerably toward recognition accuracy.

#### 4.4.7 Discussion

The work conducted by Fairfield *et al.* presented a method to recognize the color states of traffic lights that exploited a prior map of the 3D traffic light locations and camera images [63], which is a technique similar to that presented herein. They reported that the recognition precision and recall of their method were 99 % and 62 %, respectively. In contrast, the recognition precision of the proposed scheme exceeded 97 % for each color state (red, yellow, and green) for the daytime dataset (Table 4.6(c)). Moreover, the recognition recall was approximately 90 % when the recognition targets were

within a distance of 90 m (Fig. 4.10(a)). While simple comparisons are inadequate because of the different datasets evaluated, the proposed method roughly achieved a comparable level of recognition precision and improved recall of 1.4 times under favorable conditions.

One of the limitation of the proposed method is that the color states of the traffic lights may not be recognized if their 3D positions are not previously known. It is noted here that newly installed traffic lights are not contained in the 3D map. The solution to this problem includes a concept such as “connected vehicles”, which indicate vehicles that have an internet connection. If the information regarding a newly installed traffic light is updated on the 3D map in the corresponding server, the connected vehicles can download the latest 3D maps information and recognize the color states of the newly installed traffic lights.

Another limitation of this work is that the proposed method is inapplicable if the features of the traffic lights are not available owing to pixel value saturation resulting from strong solar backlight. This is a general limitation for all recognition methods using camera sensors. To avoid traffic accidents caused by such limitations, hierarchical connections may be installed between the recognition and vehicle control modules. For example, even if pixel value saturation occurs in a vehicle-mounted camera, the vehicle can stop behind other vehicles at intersections with the help of other modules, such as collision avoidance modules, that use alternative detection mechanisms, such as LiDAR sensors. In such cases, the collision avoidance modules should be at near-hierarchy levels to the control modules than traffic light recognition modules.

## 4.5 Interim conclusions on traffic light recognition

To demonstrate a practical verification of GPU applicability to perception tasks, this section presented a scheme to recognize traffic light states during driving. The proposed scheme consists of two main parts, namely utilizing the ego-vehicle location on 3D maps to extract the ROI images of traffic

#### 4.5. Interim conclusions on traffic light recognition

---

lights and utilizing a CNN-based recognizer that requires GPU acceleration. For comparison, recognizers using morphological and CNN-based processing were evaluated quantitatively for their recognition tendencies. The presented scheme requires 3D maps as prior information, which is unlike the case in several preceding studies that considered traffic light recognition using only camera images. Such a fusion can be considered as one of the key criteria for traffic-light-state recognition modules of high-level autonomous driving systems because the actual driving environment may be massively diverse, and the vehicle-mounted cameras are highly likely to capture various types of *noise*.

Two image datasets, i.e., daytime and sunset, were compiled during public driving experiments and used to train models and evaluate the proposed methods. The experimental results indicate that ROI extraction improves the recognition accuracies for both methods, thus providing more reliable recognition. The SSD recognition method outperformed the morphological processing method in terms of recognition precision and recall for almost all states of the evaluation datasets. The SSD recognizer achieved more than 97 % average precision (i.e., red, yellow, and green identification) under favorable conditions. Moreover, when the recognition targets were within a distance of 90 m, the SSD recognizer achieved approximately 90 % recognition recall.

The proposed recognition scheme can tolerate calibration and localization errors to a certain extent. However, when the errors are too large, the classifications tend to be incorrect. To obtain better results, a different and more accurate calibration method may be considered [64–66]. Finally, increasing the number of images in the training dataset can improve the average recognition precision and resilience to varying lighting conditions.

## Chapter 5

# Distributed Perception Architecture Using GPUs

The previous chapter has demonstrated that GPUs benefit perception tasks for autonomous driving because of their massively parallel computing power. However, some specific problems should be considered when applying GPUs to on-board autonomous driving systems. In this chapter, these problems are presented, and a possible solution is discussed.

### 5.1 Problems that may arise when applying GPUs to autonomous driving systems

As shown in Chapter 1, perception tasks are typically computationally intensive and require large quantities of data as inputs. However, their lengthy processing times are not tolerated by systems classified as level three and above because the outputs of these tasks become inputs to the following tasks (i.e., planning and control). When considering the actual deployment of autonomous driving systems, vehicle-mounted systems that perform online processing have severe limitations, including *low power consumption*. Perception tasks suffer from these limitations, and the tradeoffs between the limitations and processing speed are points of concern. Moreover, *system scaling* must also be considered; to ensure the safety of autonomous driving, multiple external sensors must be used to sense all surrounding circumstances

of ego vehicles. As the number of external sensors to eliminate blind spots increases, the data quantities to be processed linearly also increase, in addition to the required computational resources. To realize safe autonomous driving systems, tolerance to such data-quantity scaling is essential.

Many reported studies [67–72] have provided insights that promote application of GPUs to autonomous driving systems. In particular, GPU resource multitasking [70–72] is needed in such systems because there are numerous tasks that require GPU acceleration. From another perspective, introducing multiple GPUs is a possible solution to handle the system scaling problems noted above. However, the power consumed by traditional GPUs, including those assessed in Sections 3.1 and 4, is generally known to be comparatively large since these modules are designed for servers and desktops. When using GPUs in autonomous driving systems, the power consumption problem must be addressed.

## 5.2 Possible solution to the problems

Embedded oriented GPUs, including the Jetson series [73–76] supplied by NVIDIA Corp. and Mali series [77] designed by ARM Ltd., have received much attention lately because they provide massively parallel computing capacities with low power consumption compared with traditional GPUs. In particular, the Jetson series of GPUs were developed as embedded platforms for autonomous machines. Although the Jetson series includes powerful devices, none of them are individually sufficient to handle all tasks in autonomous driving systems, because large quantities of data and computational resources are required for the entire system, which may exceed the capacity of a single Jetson. However, the data and computational resources required for a host machine can be suppressed with a decentralized system that offloads parts of the perception tasks onto edge devices, such as the Jetson, and feeds solely processed results to the host. Although there are reported works [32–34, 78] on processing performances on edge devices, they are solely focused on processing times, which are delays of the internal edges (i.e., many studies tend to focus on non-system-wide performances). Therefore, there is a need to discuss problems that may occur

when introducing decentralized processing in autonomous driving, including delay caused by data transfer, throughput of the entire system, and tolerance against system scaling.

To discuss the validity of decentralized processing for autonomous driving, a prototype decentralized processing system that offloads parts of the perception tasks for autonomous driving onto edge devices was constructed, and the processing performance of the whole system, including delays caused by the decentralized processing, are explored. Based on the evaluation results, the validity of decentralized processing for autonomous driving systems is discussed. As noted above, typical autonomous driving systems consist of perception, planning, and control tasks. If the computational burden of all the tasks is concentrated on a specific unit (i.e., centralized unit) in a system, the entire system throughput may be degraded, which can cause serious traffic accidents during autonomous driving. In particular, typical perception tasks require large amounts of resources since they are computationally intensive. Therefore, the main aim of the proposed decentralized processing system is to avoid concentrating the computational burden on any specific unit of the autonomous driving system.

### **5.3 Model of a decentralized processing system for autonomous driving**

In this section, the model and assumptions of a decentralized processing system is presented for autonomous driving. To discuss the validity of the model, a prototype was constructed and evaluated, as will be detailed later. It may not be appropriate to apply decentralized processing to all perception tasks in autonomous driving systems since its validity depends on the processing content. The target types of perception tasks in this study are assumed to have certain attributes, namely “all direction sensing” and “tasks divisible into subtasks”, which have frequently emerged in the context of autonomous driving. This is because an architecture that divided perception tasks for surrounding circumstances (i.e., all direction) into subtasks and offloaded them onto decentralized units was employed.

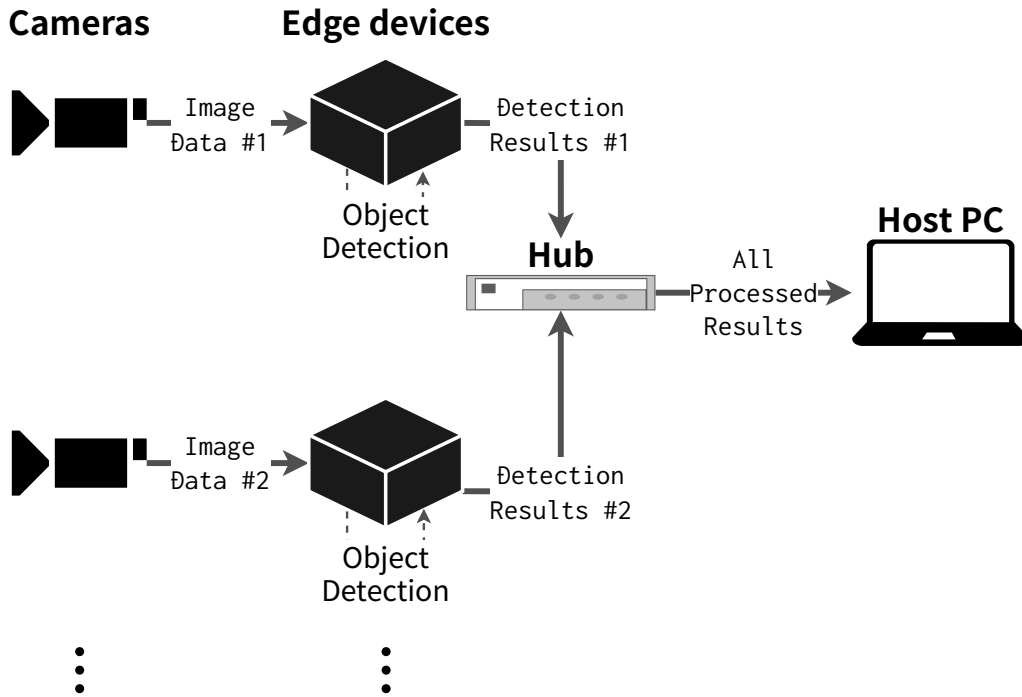


Figure 5.1: System model of decentralized processing.

As the evaluation target, a model comprising one host PC and multiple Jetsons connected to cameras individually was assumed. Every Jetson performs object detection on images captured with the connected cameras, and all the detection results are concentrated at the host PC via the network. Fig. 5.1 depicts an overview of the proposed model. As shown in Fig. 5.1, the host PC is connected to the Jetsons devices through a hub, and each Jetson is solely in charge of processing for the one specified camera. This configuration is considered to be typical for perception with a decentralized processing system. By evaluating the performance of this configuration, the validity of decentralized processing is explored. Furthermore, since the performance is vague that is required to edge devices of the perception for autonomous driving systems, exploring a rough indication of the performance requirement for such edge devices is also intended through comparing two kinds of Jetson at the later evaluation. Image-based object detection is one of the perception tasks suited to GPU acceleration. One-stage detectors proposed in recent years, including the SSD [28] and YOLOv3 [79], perform

bounding box proposal and classification in a single network and enable object detection in almost real time. These CNNs contain computationally intensive operations, and the massively parallel computational capabilities of GPUs support the fast processing of such operations. The YOLOv3-416 [79] written in TensorRT [80] was employed as the object detection task executed on each Jetson. For decentralized processing, it would be common to assign tasks according to the processing capacities and network bandwidths of each of the decentralized processing units. However, if the perception of surrounding circumstances can be divided into subtasks perceiving small regions and distributed thereafter onto decentralized units, each unit would theoretically handle an equal load. Additionally, the processing capacity and network bandwidth of each decentralized unit should be equal since a common device series (i.e., Jetson) is used. Hence, this model uniformly distributes the perception subtasks among all the decentralized units and does not consider complex task assignments.

The Jetson AGX Xavier [75] device was employed as the decentralized processing platform in this model. The Jetson series consists of a CPU, GPU, dynamic random access memory (DRAM), and power management integrated circuit (PMIC), which are all intended to run various applications with speed while consuming less power. Combining the massively parallel computational capacities of GPUs with TensorRT [80] is effective for accelerating inference processing with deep neural networks (DNNs). TensorRT is a software development kit (SDK) provided by NVIDIA Corp. that maximizes the throughput using *weights quantization*, which is an emerging technique to accelerate the inference speeds of DNNs. The Jetson series allows software configuration of the system settings, including operating frequency and number of online cores. Although users can configure these values, some presets are available; this function is called *power mode*. These presets are useful to balance computational power with power consumption because the power consumed by each Jetson device can be roughly estimated according to the usage mode. This estimated power consumption is also referred to as the *power budget*. Seven presets are available for the Jetson AGX Xavier, and Table 5.1 presents some of these.



#### 5.4. Implementation details of a prototype of decentralized processing system

Table 5.1: Part of the power-mode presets for the Jetson AGX Xavier<sup>†</sup>. The “15 W” mode is the factory default.

Property	Mode Name		
	EDP	10 W	15 W
Power Budget	n/a	10 W	15 W
Mode ID	0	1	2
Online CPUs	8	2	4
CPU Max Freq. (MHz)	2265.6	1200	1200
GPU TPCs*	4	2	4
GPU Max Freq. (MHz)	1377	520	670
DLA** cores	2	2	2
DLA Max Freq. (MHz)	1395.2	550	750
VA*** cores	2	0	1
VA Max Freq. (MHz)	1088	0	550
Memory Max Freq. (MHz)	2133	1066	1333

<sup>†</sup> Cited from [https://github.com/dusty-nv/jetson-presentations/blob/master/20181004\\_Jetson\\_AGX\\_Xavier\\_New\\_Era\\_Autonomous\\_Machines.pdf](https://github.com/dusty-nv/jetson-presentations/blob/master/20181004_Jetson_AGX_Xavier_New_Era_Autonomous_Machines.pdf)

\* TPC: Texture processor cluster

\*\* DLA: Deep-learning accelerator

\*\*\* VA: Vision accelerator

## 5.4 Implementation details of a prototype of decentralized processing system

This section presents the detailed schemes for the prototype decentralized processing system. First, the platforms used to build the decentralized processing system and their features are described. Second, the techniques for deep-learning inference on GPUs are explained with respect to accelerating the process to achieve real-time object detection. The remainder of this section presents the configurations of the decentralized processing platforms and their performance evaluations.

### 5.4.1 Robot operating system (ROS) implementation

The ROS [58] and Autoware [56, 81] were used as platforms to implement the decentralized processing system. The ROS is a middleware that provides useful tools and libraries to develop robot applications; it is well suited for decentralized processing because it employs a publisher–subscriber model to implement the interprocess communications. Owing to this characteristic, the functions of systems such as camera drivers and object detectors can be developed as modules and easily interconnected for cooperative operation. Autoware is an open-source software for autonomous driving based on the ROS; it contains function modules for perception, planning, and control in autonomous driving, including driver programs for various types of cameras. A camera driver module provided by Autoware was employed to acquire images from each of the cameras.

With regard to the ROS, the term “node” refers to an individual component module of a system, and the term “topic” refers to the actual data that are *published* (i.e., transmitted) by the nodes. Users can attach timestamps to topics programmatically. In the presented system, the camera drivers attach the current time for each published image topic. Object detection nodes inherit the timestamps of the *subscribed* (i.e., received) image topics and reattach them to the corresponding output topics. To determine the communication and processing delays, the timestamps of the topics published by the detection nodes and the instantaneous times at which the evaluation node received the topics were compared. By subtracting these timestamps from the receiving times, the delays from image acquisition to the arrival of the detection results at the host were obtained. On decentralized processing, time deviation among each unit (including the host) is the fundamental concern. For the presented system, the host and the edges should synchronize their time to maintain the consistency in timestamps of ROS topics concentrated from the edges to the host. Time consistency is essential because perception results for a frame are often considered in time-series on the host, and the host uses timestamps to identify moments to be markers, including when each image is captured and when each perception process is completed. For this reason, to synchronize the time between the

#### 5.4. Implementation details of a prototype of decentralized processing system

---

host and the Jetsons, a network time protocol (NTP) server on the host and NTP clients on each Jetson were installed.

##### 5.4.2 Deep-learning inference using TensorRT and weights quantization

As noted in Section 5.3, the YOLOv3 written in TensorRT was employed in the proposed system. TensorRT is an SDK for high-performance deep-learning inference provided by NVIDIA Corp. To improve the inference performance, the SDK adjusts the existing trained models against the GPUs on which the models are executed. Changes to the operation precision can be performed using TensorRT. Although all the inference calculations are performed in single-point floating precision (FP32) by default, some of the calculations can be replaced by half this precision (FP16) or 8-bit integer precision (INT8) via explicit specification during execution. If the operation precision is changed to FP16 or INT8, then TensorRT generates an adjusted-precision inference engine; using the generated engine for inference is expected to improve the throughput and decrease resource consumption [80]. Since NVIDIA Corp. has recently released some GPUs that contain exclusive cores to process INT8 operations, the operational throughput may be significantly improved and resource consumption may be degraded solely by shifting to the INT8 operation precision. Because the available value range and granularity are quite different between the single-point floating and 8-bit integer precisions, *calibrations* are required to generate the INT8 inference engine. During the calibration process, TensorRT executes an FP32 inference engine for multiple inputs to explore the internal value range of the FP32 engine and scale it to that of the 8-bit integer range. The amount of resource consumption and detection performance in the cases of the FP32 and INT8 inference engines were evaluated. To generate the INT8 inference engine, the 2017 validation image set [82] of MS COCO [25] was used for calibration.

### 5.4.3 Jetson settings

To explore the potential capabilities of the proposed system, the minimum delays that could be achieved when operating the Jetsons at full capacity were measured. Additionally, changes in the power consumption for changes in the operation precision were measured. To this end, the power budget of each Jetson was set to “n/a” (i.e., mode name “EDP” , as shown in Table 5.1); then, the effects of power budget settings on the object detection task performance are shown in Section 5.5.5.

## 5.5 Performance analysis

To quantify the performance of the proposed decentralized processing system for environmental perception during autonomous driving, the following items were evaluated:

1. Delays based on decentralized processing
2. Effects of weights quantization on resource and power consumptions
3. Limitations of centralized processing
4. Effects of weights quantization on detection performances of different series of edge devices

### 5.5.1 Experimental setup

The following setups were deployed to evaluate the system.

#### Host PC (centralized environment)

- CPU: Intel core i7-8750H @ 2.20GHz (12 cores/24 threads)
- Memory: 16 GB
- GPU: NVIDIA GeForce GTX 1060M (1280 CUDA cores, 6 GB memory)
- Networking: 10/100/1000 BASE-T Ethernet

### **Jetson AGX Xavier (decentralized environment)**

- CPU: ARM v8.2 64-bit
- Memory: 16 GB
- GPU: NVIDIA Volta GPU (512 CUDA cores with 64 tensor cores)
- Networking: 10/100/1000 BASE-T Ethernet

### **Jetson Nano (for comparison)**

- CPU: ARM A57 64-bit
- Memory: 4 GB
- GPU: NVIDIA Maxwell GPU (128 CUDA cores)

### **Cameras**

- Device: Blackfly S GigE from FLIR Systems, Inc.
- Frame rate (set to): 20 fps
- Image size (set to): 1280(W) × 960(H) × 3(bytes/pixel)

### **Hub**

- NETGEAR GS108PEv3
- Number of 10/100/1000 Base-T RJ45 ports: 8 (4 PoE 802.3af Ports included)

### **Object detection results**

- Self-defined ROS topic
- Approximately 1 KB per detected object, including bounding box, detection score, and detected object category

## **5.5.2 Delays based on decentralized processing**

Fig. 5.2 depicts the connection diagram for the evaluations. The measured delays are as follows:

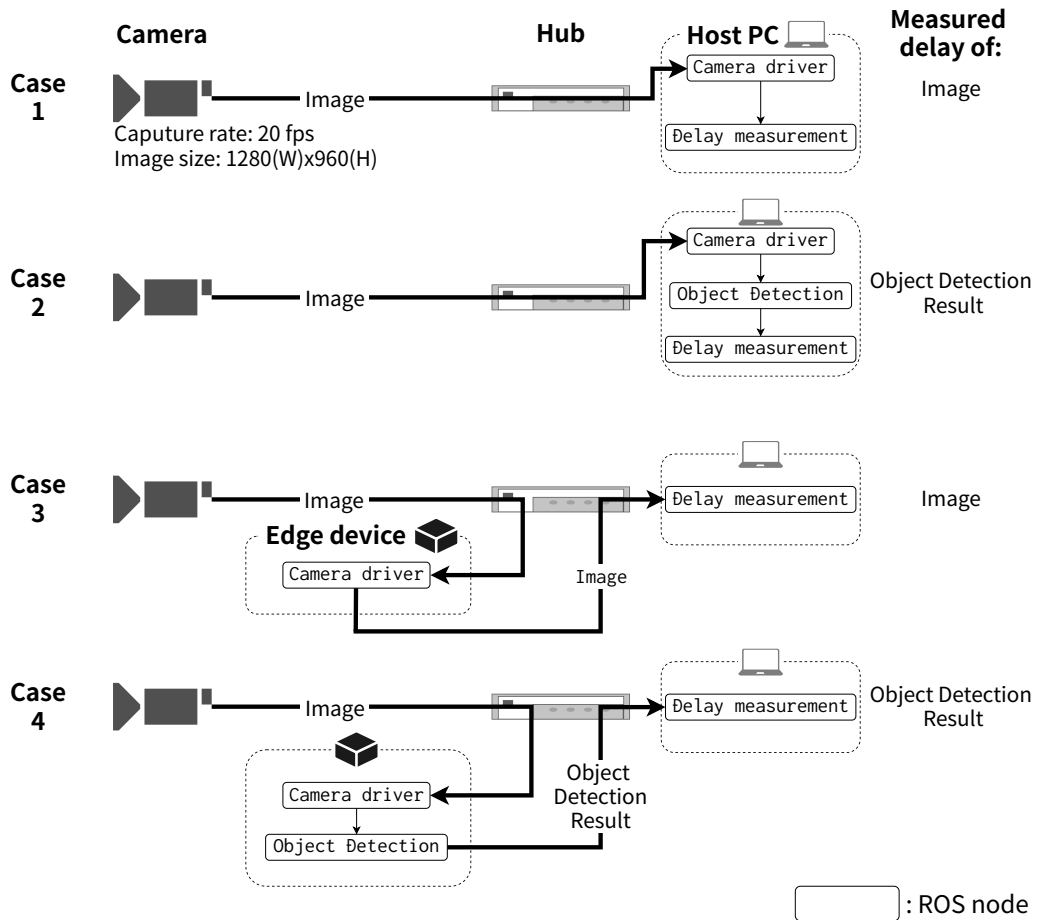


Figure 5.2: Connection diagram to evaluate data delays on decentralized processing. In cases 3 and 4, the camera driver and object detection modules were offloaded to the edge devices to measure the effects of decentralized processing.

**Case 1:** Elapsed time until the host PC subscribes the images published by the camera driver node running on the same host.

**Case 2:** Elapsed time until the host PC subscribes the results published by the object detection node running on the same host. A camera driver node was also running on the same host in this case.

**Case 3:** Elapsed time until the host PC subscribes the images published by the camera driver node running on the edge device.

**Case 4:** Elapsed time until the host PC subscribes the results published by the object detection node running on the edge device. A camera driver node was running on the edge device in this case.

During evaluations, the capture rate of each camera was set to 20 fps; this capture rate is assumed to be sufficient for autonomous driving systems because some kinds of sensor devices, such as the 360° LiDAR units, operate at 10 fps, and the systems must work in synchrony with such devices. To measure the individual delays, the differences between the timestamps of the target topics (i.e., the time at which the image data were fed to the system by the driver node) and the moment at which the host PC received these topics were considered, as described in Section 5.4.1. Since the implementations were on the ROSs, the image data were not available to the other functions (i.e., the other ROS nodes) until the camera driver node fed the data to the ROS network. Hence, the measured delays represent the time elapsed between the instants at which the image data were available and the instants at which the desired data reached the host. Fig. 5.3 indicates the delays measured for each case of Fig. 5.2. In Case 1, where the image data were published and subscribed inside the host, the average delay was approximately 1.47 ms. In Case 3, where the image data were published by the edge device and subscribed by the host, the average delay was approximately 10.4 ms. These correspond to the image transfer delays when solely the camera driver was offloaded to the edge device (i.e., decentralized unit). The simple offloading of the camera driver was observed to increase the delay by approximately ten times. Since systems using decentralized processing sometimes suffer from such delays, system-wide

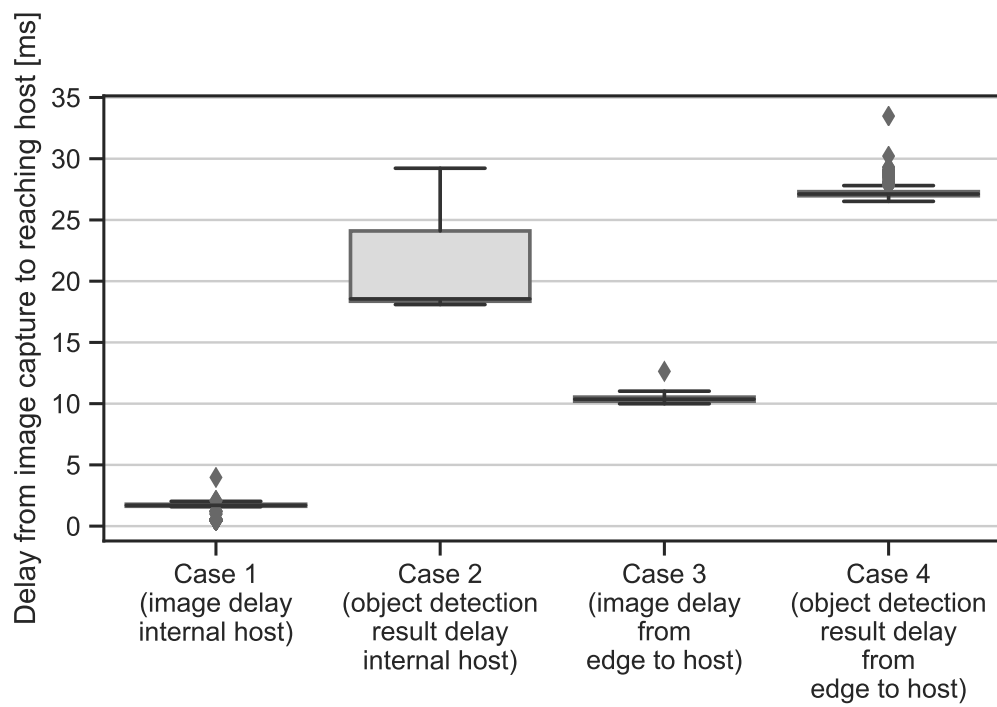


Figure 5.3: Delay comparisons with and without decentralized processing.



evaluations should be considered to benefit from decentralized processing. When image publishing and object detection were performed on a single host, i.e., Case 2, the measured average delay was approximately 21.0 ms. This delay is assumed to roughly correspond to the time elapsed for object detection. However, as the exact elapsed time for object detection varies depending on the GPU, Case 4 was considered to measure the total delay when offloading the camera driver and object detection modules to the edge device. When image publishing and object detection were performed on the edge device (Case 4), an average delay of approximately 27.0 ms was observed for the detection results to reach the host. Although the largest delay was observed in Case 4, the measurements revealed that the object detection results could reach the host without frame drops even with offloading to the decentralized edge devices when the camera capture rate was 20 fps (=50 ms/frame).

### 5.5.3 Effects of weights quantization on resource and power consumptions

Fig. 5.4 depicts the consumption of GPU computational resources on each platform when the FP32 and INT8 inference engines were used. The left side of the figure shows the measurement results for a GPU on the host, and the right side shows the results for a GPU on the edge device. In the host, although approximately 46.8% of the resources were occupied when the FP32 inference engine was in operation, this utilization decreased to approximately 27.8% when the INT8 inference engine was in operation. In the edge device, these average usages were approximately 86.3% and 34.4% for the FP32 and INT8 inference engines, respectively. As shown in Fig. 5.4, the quantized network decreased consumption of the GPU computational resources during inference on both the host and edge device. Especially on the edge device, the average GPU computational resources consumed by the INT8 inference engine was approximately 2.51 times lower than that consumed by the FP32 inference engine. The consumption rates were different between the host and the edge device. This might partially result from the difference in the number of cores in each GPU; however, this suggests that TensorRT

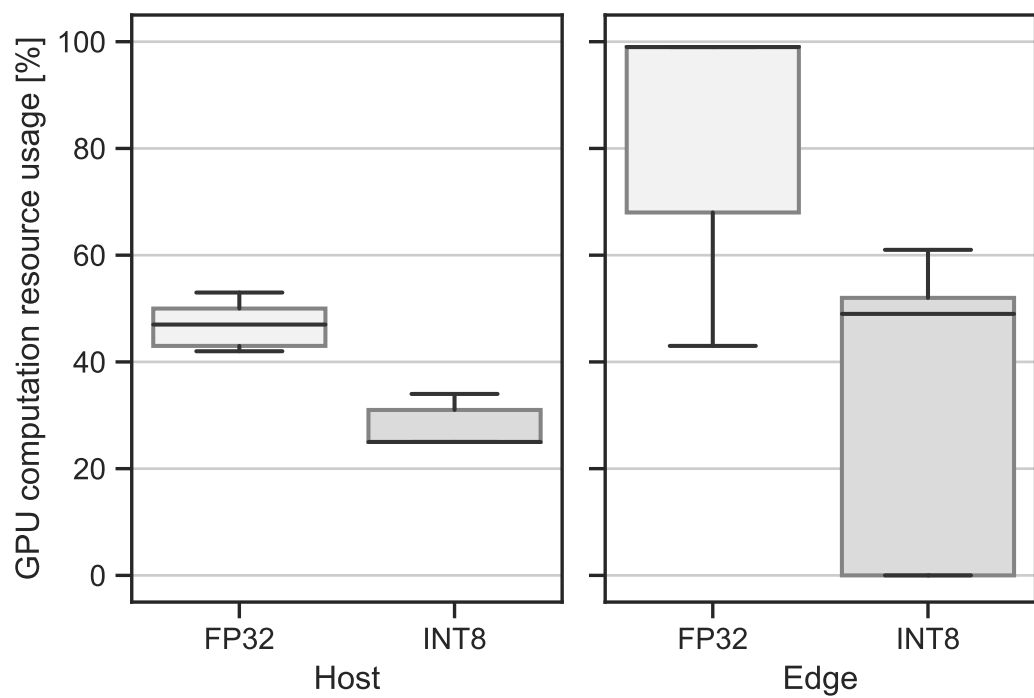


Figure 5.4: GPU computational resource utilization for different operation precisions wherein object detections are executed on the host (left) and edge device (right).

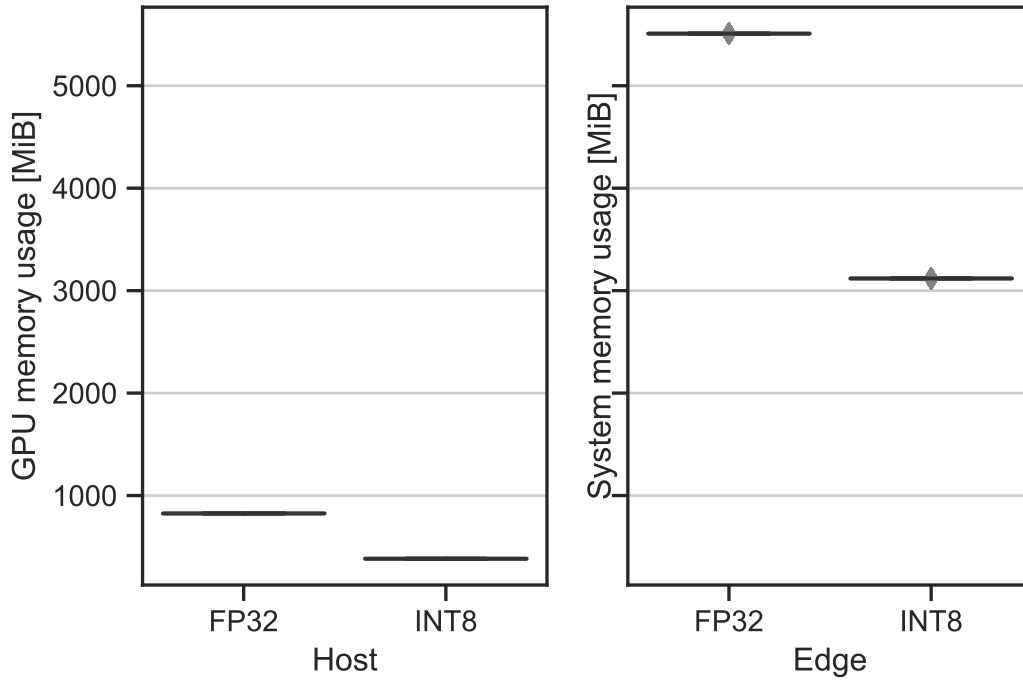


Figure 5.5: GPU memory utilization for different operation precisions measured during the experiments in Fig. 5.4.

performed different adjustments for each GPU on the host and edge device when generating inference engines.

Fig. 5.5 indicates the memory consumed during the experiments in Fig. 5.4. Note that the measurements for the host (left side of the figure) indicate consumption of GPU memory, whereas the measurements for the edge device (right side of the figure) indicate the total system memory consumption. This is because there is no interface to measure solely the GPU memory consumption on the edge device. Reusing GPU memory region once allocated is generally recommended to maximize performance because GPU memory allocation is an expensive operation [83]. The standard deviations of memory consumption observed for both GPUs were nearly zero. A possible reason for is that the memory regions were reused in the engines generated by TensorRT. Similar to the computational resources, the memory consumptions degraded on both GPUs when the network weights were quantized. Specifically, on the edge device, the average consumption

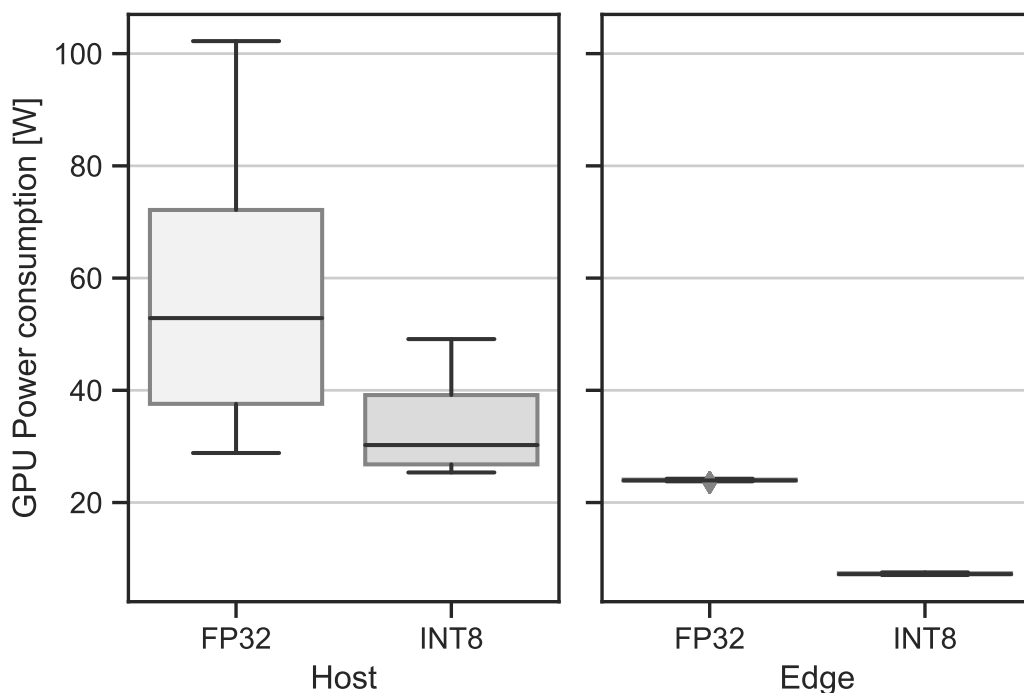


Figure 5.6: Power consumed by the GPU for different operation precisions measured during the experiments in Fig. 5.4.

reduced by approximately 1.77 times.

The power consumed by the GPUs for the experiments in Fig. 5.4 is illustrated in Fig. 5.6. The same trends of the effects of weights quantization for power consumption were observed: it decreased on both GPUs. The lowest average power consumption was 7.31 W, which was achieved with a combination of edge processing and inference with the INT8 engine.

#### 5.5.4 Limitations of centralized processing

To identify the limitations of centralized processing, the ROS topic (i.e., object detection results) rates were measured for different numbers (one to six) of running object detection nodes on the host. These measurements were performed with the settings of Case 2 in Fig. 5.2. To simulate a situation wherein a single host processed data from multiple cameras, multiple object detection nodes subscribed images published by a single camera driver node

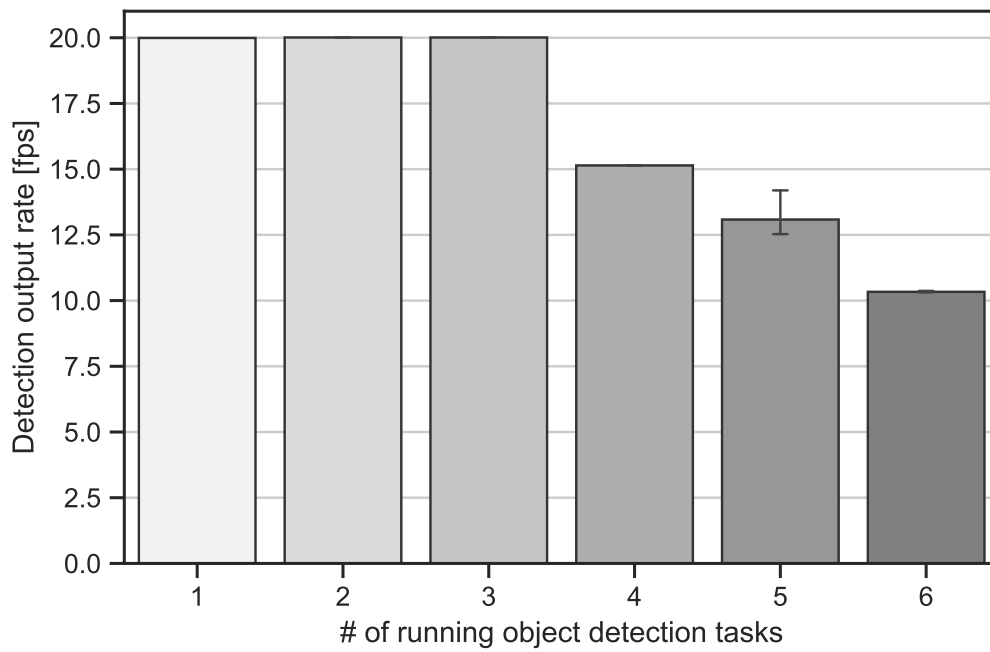


Figure 5.7: Output rates of object detection results in the centralized environment.

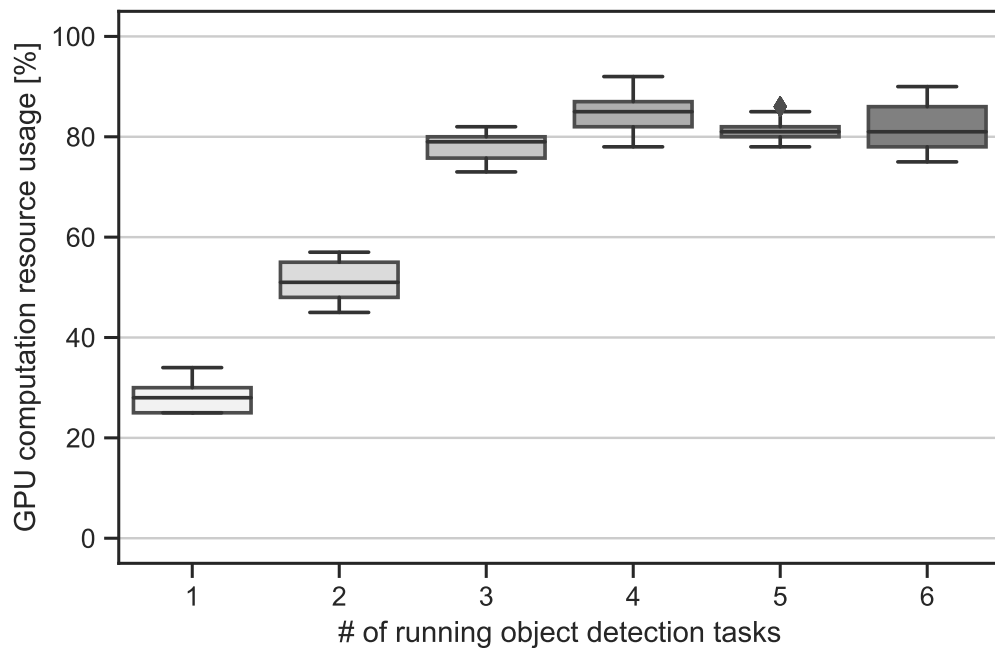


Figure 5.8: Utilization of GPU computational resources on the host for different numbers of tasks.

in these measurements. Fig. 5.7 indicates these measurement results; when the number of object detection nodes exceeded three, the topic rate was lower than 20 fps. Because the camera capture rate was set to 20 fps, frame dropping occurred. Fig. 5.8 demonstrates the consumption of GPU computational resources in this experiment. As shown in Fig. 5.7, frame dropping is observed when 4—6 nodes are operating, and the average GPU resource consumption is over 80 % for these situations. This result suggests that centralized processing could decrease the performances of perception tasks if it is in charge of multiple sensors.

### 5.5.5 Effects of weights quantization on detection performances of different series of edge devices

The effects of changes in operation precision of the network on the detection accuracy and speed are presented in this section. Additionally, the execution times for object detection with different devices from the Jetson series are compared later. Although the Jetson Nano has a smaller body and lower power consumption, the Jetson AGX Xavier is superior in terms of computational power. By comparing these two, a more appropriate one for the edge devices for autonomous driving systems is explored.

Table 5.2 indicates the average precision and recall values for object detection achieved by the FP32 and INT8 inference engines. The MS COCO 2014 validation image set [84] was used to calculate the average precision and recall values. To simulate the environmental perception of autonomous driving, data from six specific categories (*car*, *pedestrian*, *bus*, *truck*, *bicycle*, *motorcycle*) contained in the dataset were used to calculate the metrics. The value of each item was slightly lowered after changing the operation precision from FP32 to INT8. The SSD512 [28], which is another state-of-the-art detector, achieved  $mAP-50 = 0.485$  for the MS COCO 2015 test image set. Although a direct comparison is not fair because the datasets used were different, the INT8 inference engine achieved a detection accuracy comparable to that of the other state-of-the-art detector.

Fig. 5.9 illustrates the inference times of object detection achieved by each power mode preset using inference engines with different operation

Table 5.2: Precision and recall comparisons for different operation precisions for six specific categories on the MS COCO val 2014 dataset.

Metrics	IoU range <sup>(1)</sup>	area <sup>(2)</sup>	maxDets <sup>(3)</sup>	FP32	INT8
Average Precision	0.50	all	100	<b>0.552</b>	<b>0.498</b>
	0.75	all	100	0.373	0.353
	0.50:0.95	small	100	0.098	0.069
	0.50:0.95	medium	100	0.343	0.304
	0.50:0.95	large	100	0.587	0.583
Average Recall	0.50:0.95	all	1	0.237	0.226
	0.50:0.95	all	10	0.372	0.342
	0.50:0.95	all	100	0.375	0.344
	0.50:0.95	small	100	0.114	0.077
	0.50:0.95	medium	100	0.381	0.335
	0.50:0.95	large	100	0.645	0.635

<sup>(1)</sup> IoU: Intersection over union

<sup>(2)</sup> According to <https://cocodataset.org/#detection-eval>, the definitions of object area sizes are as follows: small:  $\text{area} < 32^2$  pixels, medium:  $32^2 < \text{area} < 96^2$  pixels, large:  $96^2 \text{ pixels} < \text{area}$

<sup>(3)</sup> maxDets: thresholds on maximum detections per image



## 5.5. Performance analysis

---

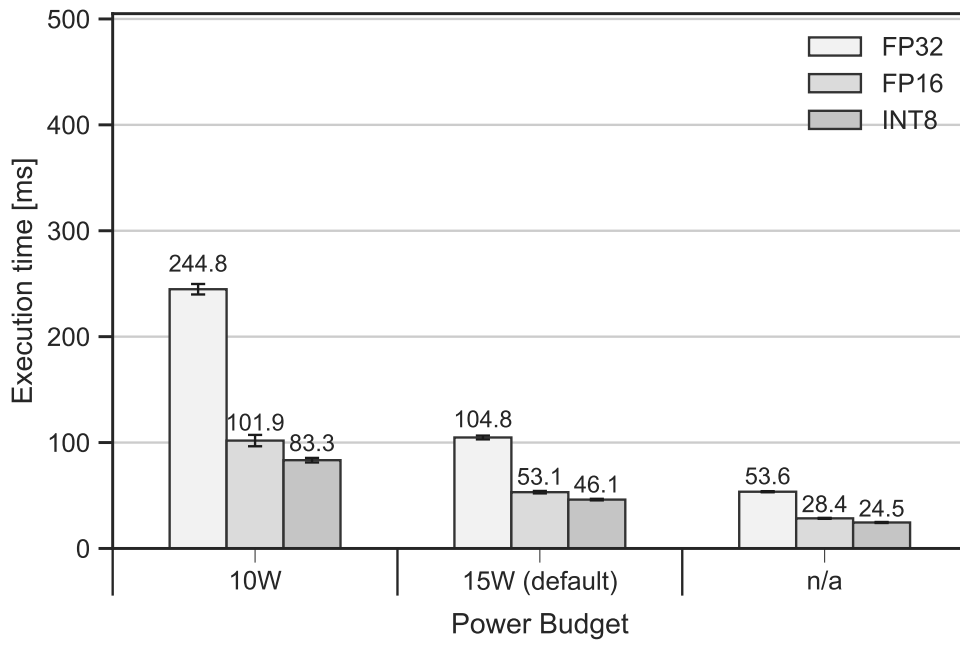


Figure 5.9: Execution time comparisons for power budget and operation precision on the Jetson AGX Xavier.

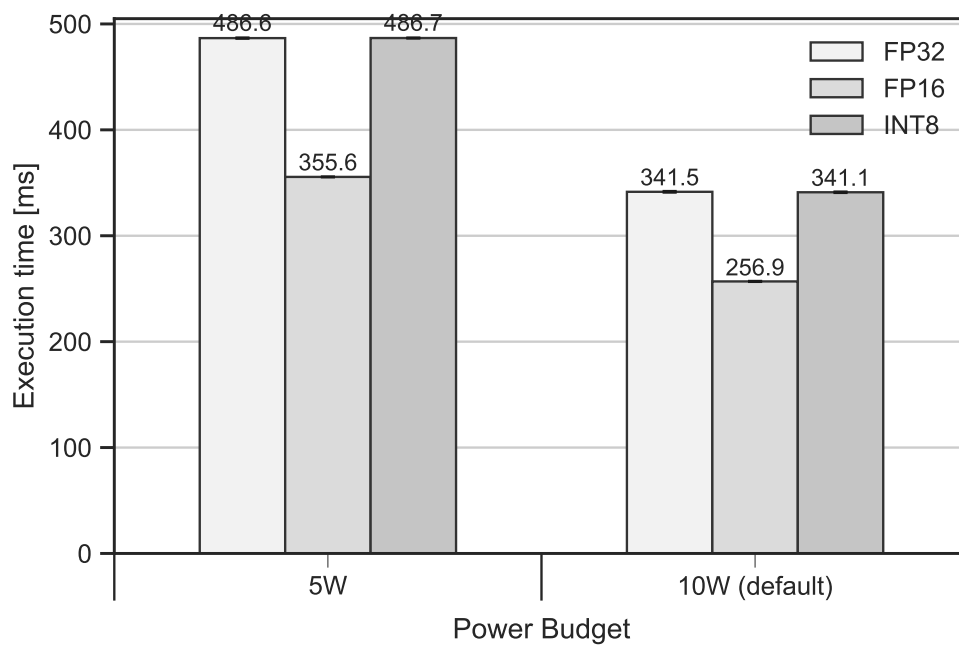


Figure 5.10: Execution time comparisons for power budget and operation precision on the Jetson Nano.

precisions. With the help of tensor cores and deep-learning accelerators (DLAs), which can rapidly process FP16 and INT8 operations, the execution time for each power mode preset was significantly improved by changing the operation precision. Furthermore, the execution time differences between the power mode presets were also significant. Therefore, the differences in the maximum frequencies of the GPUs and DLAs for each of the power mode presets mainly caused the performance differences. Since the “EDP” mode (i.e., the mode with power budget “n/a”) places no restrictions on the Jetson resources, results labeled “n/a” in Fig. 5.9 can be seen as maximum performances that this device can achieve. The power budget “n/a” with the INT8 inference engine achieved an average inference time of approximately 24.5 ms ( $\approx 40.8$  fps). Note that evaluation results until here were measured under the “EDP” mode if without being noted, therefore the presented configuration achieved low power consumption illustrated in Fig. 5.6 as well as fast processing. For comparison, similar measurements were obtained using the Jetson Nano [76]; the results are demonstrated in Fig. 5.10. Two power-mode presets are provided for the Jetson Nano, and the power budgets for these presets are 5 W and 10 W, respectively (10 W is the factory default). Because the Jetson Nano contains four-times fewer CUDA cores than the Jetson AGX Xavier, the overall processing speed was lower as well. Moreover, changing the operation precision did not improve the inference speed significantly. This is because the GPUs on the Jetson Nano are based on the Maxwell architecture, which does not contain any tensor cores or DLAs. From the viewpoint of autonomous driving, an execution time of approximately 260 ms ( $\approx 3.8$  fps), which is the highest processing speed achieved by the Jetson Nano, cannot be accepted as an on-board processing delay. Through comparisons, it was concluded that the Jetson AGX Xavier is preferable to the Nano at this point.

# Chapter 6

## Discussion

### 6.1 GPU-accelerated perception algorithms

Based on the studies presented in Chapter 3, it can be inferred that GPUs are effective for autonomous driving perception tasks from the perspective of computational acceleration.

Object detection algorithms are rapidly progressing, and the employment of deep-learning-based object detectors is inevitable because they typically outperform traditional pattern recognition algorithms in terms of execution speed and accuracy. However, it is noteworthy that deep-learning-based object detectors basically perform statistical inferences based on training data. Therefore, the quantity and variety of training data directly affects the generalization capabilities of detection. Because the conditions of the driving environment, including visual scenes and lighting, are largely diverse given limited training data, proving tests are vital toward the actual deployment of autonomous driving systems.

Another consideration is the reliability and interpretability (transparency) of the deep-learning-based method. Because deep-learning methods are typically end-to-end approaches (in other words, they directly output information in the desired format from the input), they are often used as black-box modules, in which the relationships (reasons) between the model inputs and outputs are unclear. This prevents developers from identifying error causes during the development phase and renders it difficult

to measure the reliability of a system during deployment. From the perspective of the interpretability of the inference process, traditional pattern recognition or rule-based methods are often superior to deep-learning-based methods. Depending on the case, such as when a target can be described in terms of strict rules, end-to-end methods and their combinations with rule-based inference (e.g., inter-frame filters for traffic-light-state transitions, as mentioned in Section 4.3.3) should be considered. The interpretability of deep-learning-based methods is gradually gaining importance, and considerable relevant research efforts are in progress [29–31]. Measuring the system reliability is one of the key for autonomous driving systems at levels three and above because such systems must address the unreliable states of the systems (passing back vehicle control to the driver at level three or managing with the available system at levels four and five). Therefore, techniques to improve the interpretability of deep-learning-based methods are critical for autonomous driving.

## 6.2 Decentralized processing using embedded-oriented GPUs

This section presents the validity of decentralized processing for environmental perception tasks in autonomous driving.

As shown in the experimental setup (see Section 5.5.1), the image size was set to 1280(W)  $\times$  960(H), and the frame rate was set to 20 fps for the camera configuration. Moreover, a self-defined ROS topic was deployed to transfer the object detection results of approximately 1 KB per object. Briefly, the amount of data input to the system by a single camera was  $1280 \text{ (W)} \times 960 \text{ (H)} \times 3 \text{ (bytes/pixel)} \times 20 \text{ (fps)} = 73728000 \approx 70 \text{ MB/s}$  under this configuration. By contrast, when only the object detection results were transferred, the amount of data decreased by a few orders of magnitude and was  $1 \times N \text{ (objects/frame)} \times 20 \text{ (fps)} \approx 20N \text{ KB/s}$ . A comparison of the amount of data transferred is presented in Table 6.1. Considering the network load of the host machine, the decentralized processing was more tolerant to system scaling because the amount of data increased linearly based on the

Table 6.1: Comparison of network traffic amounts with and without decentralized processing under experimental settings.

Processing type	Breakdown				approx. Total (MB/s)
	Height (pixel)	Width (pixel)	byte / pixel	fps	
w/o decentralized (i.e., centralized)	960	1280	3	20	70
w/ decentralized	data-size / object (KB)		objects / frame	fps	0.02N
	1		N*	20	

\* Because the number of detected objects per frame is not fixed, this N represents a dynamic value.

number of external sensors.

To recognize the circumstances surrounding the ego vehicle, the system was assumed to have multiple cameras equipped with standard field of view (FOV) lenses that detected objects by processing the captured images. Typically, a standard FOV camera lens has a view angle of  $25^{\circ}$ – $50^{\circ}$ . In such cases, at least six to seven cameras are required to capture all the directions. Wide FOV cameras, including fisheye cameras, may alternatively be used to capture the surroundings using fewer cameras. The authors of [85] published a dataset for autonomous driving that included images captured using fisheye cameras and reported a baseline object detection experiment using the faster R-CNN [43] and fisheye-camera images, where the detection accuracies were significantly lower than those achieved using other datasets. It was observed that accurate object detection using fisheye-camera images might be difficult owing to significant lens distortions (*“the orientation of objects in the periphery of images being very different from central region”*). For autonomous driving perception systems, one-stage detectors, including

YOLOv3, may be preferable to two-stage detectors, including the faster R-CNN, because fast and online object detection is required. Meanwhile, one-stage detectors typically suffer from lower accuracies of bounding box regressions compared with two-stage detectors [44]. Hence, it is suggested that the accuracies of one-stage detectors may be reduced if wide-FOV images with large distortions are applied as inputs. As such, processing images containing slight distortions captured using multiple cameras may be a promising approach for accurately detecting objects in the surroundings.

If a single machine is responsible for all the cameras, then frame dropping may be an issue because of the heavy burden of data transfer and computation. Assuming that a centralized system comprising a single host machine responsible for six cameras is set to the same configurations as the experimental setup, the transferred image data can theoretically occupy a constant 420 MB/s of the system network. Such a system will suffer from frame dropping, as shown in Fig. 5.7, as well as the high occupancy rate of the network. By contrast, if decentralized units are introduced to each camera, then the network burden decreases to  $0.12N$  (MB/s), where  $N$  is the number of detected objects per frame and is typically under 100 even for images captured while driving in an urban area with complex surroundings. With an additional assumption of  $N=100$  (objects/frame), the network burden would be 12 MB/s, which is 35 times lower than that of the centralized system. The host machine experiences little computational burden for object detection because it is offloaded to the decentralized units, and the other tasks can safely utilize the host resources. With respect to autonomous driving, frame dropping and resource occupation by a single task prevent the appropriate operation of the entire system and may cause serious accidents. Therefore, using multiple cameras to capture the surroundings and processing images using decentralized computational resources instead of a single machine may be desirable; these approaches are expected to be a realistic for actual deployment in autonomous driving systems.

The increased power consumption caused by the decentralization of computational resources remains a concern. As shown in Fig. 5.6, the power consumption of the GPU can be decreased using an embedded GPU and network weights quantization. It is noteworthy that the values presented

herein indicate the power consumption of the GPU exclusively and do not include those of other components that may consume excess power, such as DRAM. As described in Section 5.4.3, no limitations were set with respect to the power budget of the edge device in the experiments detailed in Section 5.5.2. The power consumption of the entire system is assumed to become lower when limitations were set for the power budget because the power budget affects the configuration for the whole components of the edge device, including maximum frequencies of CPU/GPU/memory and the number of online cores. For example, as shown in Fig. 5.9, the average inference time was approximately 46.1 ms ( $\approx 21.7$  fps) in the case involving a 15 W power budget (factory default for the Jetson AGX Xavier) and an INT8 inference engine. Hence, it is assumed that the inference process itself can exceed 20 fps even with the limitation of the power budget. Meanwhile, as shown in Figs. 5.4–5.6, using the INT8 inference engine decreases the consumption of computational resources, memory, and power; moreover, a maximal operation of approximately 40 fps was achieved. As an alternative to set limitations on the power budget, using a single edge device to manage multiple (a few) cameras may also suppress the total power consumption per camera.

Increased system complexity is another concern in decentralized processing. In general, decentralized processing increases the system’s complexity, which occasionally results in the deterioration of the system’s reliability (i.e., the ability of the system to function without failure). The software complexity of the proposed system did not increase significantly with the introduction of decentralized processing because of the ROS; however, the hardware complexity increased, which was in fact inevitable. The robustness of the system against failure should also be considered. For autonomous driving systems at levels four and above, system failure during driving must be managed by the systems themselves (e.g., stopping the car in a safe place) as the driving tasks are not to depend on human drivers [2]. In centralized processing systems, the margins of the computational resources to manage failure are estimated to be limited because a single computational unit must address all processing requirements. By contrast, the resource margins are considered to be relatively sufficient if a decentralized processing approach



is employed because the computational burden can be prevented from being concentrated on a specific section of the system. Hence, a decentralized system can offer higher safety against failure than a centralized system.

One of the limitations of this study with regard to the prototype decentralized processing system is that driving at high velocities, such as driving on highways with no velocity limits, was not considered. Because the prototype considered only the velocity limit for driving in urban areas, such as 60 km/h, the camera frame rate was set to 20 fps in the experimental setup; this configuration is not adequate for driving at high velocities. More specifically, including the camera frame rate, network protocol, and ROS topic (i.e., transferred data) format, should be considered from the perspective of processing speed to relax this limitation. However, the advantages of decentralized processing, including reduction in power consumption and non-concentration of the computational burden, can be utilized to construct autonomous driving systems.

The price of GPUs applied as components of autonomous driving systems is another concern. In fact, the costs of GPUs employed in this study were not trivial; however, the price of GPUs should decrease as the application of GPUs widens. With regard to LiDARs, a typical external sensor for the environmental perception of autonomous driving systems, some manufacturers offer LiDARs at relatively affordable costs, such as under \$500. A possible cause for this price decline is the increase in demand for LiDARs in the autonomous driving domain. A similar price decline is hoped for GPUs (particularly for embedded-oriented ones) along with an increase in their application; it might be desirable that the price range of future devices are affordable, i.e., similar to that of the Jetson Nano (currently around \$200), while affording computational power comparable to that of the Jetson AGX Xavier. Moreover, the same level of computational power tends to be offered at a lower price in the future (e.g., the same or higher level computational power of supercomputers a few decades ago can currently be obtained at an affordable price). Hence, the price of GPUs was not regarded as a concern or limitation in this study.

# Chapter 7

## Conclusion

### 7.1 Summary of contributions

This dissertation presents methods to accelerate perception tasks for autonomous driving using GPUs to fulfill the criteria of high-level autonomous driving systems. By exploring three current research topics, this study aims to validate GPU usage for autonomous driving.

Object detection is a typical perception task, which is a task of identifying objects, such as vehicles or pedestrians, from raw or preprocessed sensor data. Although image-based object detection is vital to autonomous driving systems, its high computational cost prevents its practical usage, and this tradeoff between detection accuracy and computational cost is one of the primary problems to be addressed in object detection tasks. In Chapter 3, the GPU implementation schemes for traditional object detection tasks are presented. The performance improvements achieved using GPUs as well as detailed quantitative evaluations with different setups to accelerate traditional object detection algorithms on GPUs are presented. The implementation schemes were based on massively parallel computing threads using event streams and texture memory. In the best-case scenario, the proposed GPU implementation achieved an  $8.6\times$  performance improvement compared with a high-end CPU implementation, while maintaining the detection accuracy.

Recognizing traffic light states is another perception task in autonomous

driving. Image-based methods are the first option to manage this task because traffic lights are designed for the visual perception of humans; however, images captured via vehicle-installed cameras typically contain various objects that are unassociated with traffic light state recognition, thereby causing misrecognition. Hence, leveraging high definition maps as prior knowledge for the driving environment is a promising approach. The scheme for recognizing traffic light states from images is reviewed in Chapter 4. The scheme comprises two main aspects: (i) utilizing ego-vehicle locations on 3D high definition maps to extract ROI images of traffic lights; (ii) utilizing a CNN-based recognizer that requires GPU acceleration. ROI image extraction leveraging high definition maps significantly reduced unrelated objects for traffic light recognition and enabled the CNN-based recognizer to exhibit its recognition performance; furthermore, it was revealed that the GPUs resulted in a  $130\times$  faster execution of the CNN-based recognizer. Under favorable conditions, this scheme achieved an average precision exceeding 97 % for recognizing traffic light states using data acquired during public driving experiments. Moreover, if the recognition targets were within 90 m, then a maximum recognition recall of approximately 90 % could be achieved.

Computation offloading and system scaling are also concerns in the actual deployment of autonomous driving systems because the concentration of computational and/or data transfer prevents the system from operating as intended. In Chapter 5, computation offloading and system scaling problems were explored. Furthermore, as a possible solution to these problems, a model of a decentralized processing system that utilizes embedded-oriented GPUs to fulfill the criteria of high-level autonomous driving systems was presented. Additionally, the validity of the decentralized processing model was discussed. To clarify the effects of introducing decentralized processing, including the delays caused by the data transfer and throughput of the entire system, a prototype system that performs object detection on embedded-oriented GPUs was constructed and evaluated. Quantitative evaluations showed an average delay of approximately 27 ms between feeding an image to the system and observing object detection results at the host, indicating that even though the measured delay included overheads from decentralized processing, frame dropping did not occur during the detection

task when the camera capture rate was approximately 20 fps. In addition to computational offloading, the experimental evaluations demonstrated that the network load of the host can be reduced by several orders of magnitude. Based on the quantitative evaluations, it can be concluded that autonomous driving systems can reap the benefits of decentralized processing, even with the resulting delays.

## 7.2 Future direction

In this study, the potential of GPU acceleration of computations for autonomous driving was explored. Although GPUs should benefit perception tasks for autonomous driving, the perception accuracy depends on the algorithms because the methods investigated to apply GPUs in this study do not change the basic algorithm flow. Therefore, identifying methods to improve the perception accuracy is essential for expanding the applicable area of autonomous driving. Such an improvement may be achieved using a single perception algorithm or a sophisticated pipeline comprising subdivided tasks. In this regard, deep-learning-based methods are highly promising; the interpretability of those methods (i.e., *explainable AI* or *XAI*) is crucial for obtaining social acceptance as well as further accuracy improvements.

Regarding accelerating computations using GPUs, the derivation of the potential capacities of GPUs as well as the evolution of GPUs should be considered. Because one of the key features for achieving fast computation on GPUs is to hide memory access latency by processing executable threads simultaneously, the number of CUDA threads per unit (i.e., warp) is a typical parameter to be adjusted to derive the GPU's performance. As another example, DLA, which is a dedicated unit for processing deep-learning-related computations efficiently, has recently been installed on GPUs as a result of widespread deep-learning usage. As shown by the evaluation results in Chapter 5, leveraging such GPU capacities is essential to fulfill the criteria of autonomous driving systems, such as fast computation and low power consumption.

Owing to the steady development of the performance of GPUs over the past decade, further considerations are necessitated prior to their practical

deployment. For example, this study demonstrated that a combination of embedded-oriented GPUs and network weights quantization can reduce power consumption considerably, and that object detection inference can be achieved with approximately 7.3 W per camera on average (see Chapter 5). However, further power consumption reductions may be necessitated because of the significant number of components that require electrical power in high-level autonomous driving systems. In such situations, setting limitations on the power consumed by decentralized units is one of the viable countermeasures. Because power consumption and computational capacity generally exhibits a tradeoff relationship, power limitations are necessitated to balance performance with other considerations. Additionally, verification tests have gradually become vital for gaining social acceptance toward autonomous driving. Hence, the continuous investigation of practical settings and verification/validation tests may encourage the development and acceptance of high-level autonomous driving technologies.

I hope that the findings presented herein can facilitate the acceleration of task computations while satisfying the criteria for the implementation of high-level autonomous driving systems in the future.

# Acknowledgment

I am profoundly indebted to my adviser, Associate Professor Shinpei Kato (currently with the University of Tokyo), for his guidance, understanding, generosity, and most importantly, sincerity, during my studies at Nagoya University. His mentorship, uncompromising stance for research, and encouragement throughout this work helped me not only grow as a researcher but also make personal progress.

I would like to express my deep respects and gratitude to Professor Masato Edahiro for his invaluable suggestions and directions throughout this work. His insightful indications and comments on the work fostered my ability to see things from a different standpoint. It was my great pleasure to be given numerous opportunities to work in the fields of GPGPUs and autonomous driving.

I am very grateful to Professor Hiroaki Takada for his understanding and generosity. I wish to sincerely express my respect and appreciation for his insightful comments and advice on my studies.

I would like to thank Professor Shinya Honda (currently with Nanzan University) for providing valuable comments and insights during my studies.

I would also like to express my special thanks to the past and present members of Edahiro and Takada Laboratory. Their helpful comments and cooperation were encouraging and stimulating to me, and the fulfilling discussions helped me develop new ideas with respect to my studies. I am additionally thankful to them for making my academic life quite enjoyable.

Outside the work environment, I wish to acknowledge and thank my excellent friends, who have always been cheerful and supportive.

Last but not the least, I thank my family for their steady and warm support, encouragement, and understanding of me.

# Bibliography

- [1] Cabinet Office Japan. *Annual Report on the Aging Society [Japanese full version] FY 2014*. Japanese Government, 2014.
- [2] SAE On-Road Automated Driving Committee and others. SAE J3016. Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles. Technical report, SAE International, 2016.
- [3] JSAE Standardization Board. JASO TP 18004. Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles. Technical report, Society of Automotive Engineers of Japan, Inc., 2018.
- [4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [5] NVIDIA. CUDA Toolkit Documentation. <https://docs.nvidia.com/cuda>, 2020.
- [6] S. Rennich. CUDA C/C++ Streams and Concurrency. <http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf>.
- [7] N. Wilt. *THE CUDA HANDBOOK : A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013.

- [8] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *Proc. of the IEEE conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 2018.
- [9] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12697–12705, 2019.
- [10] Tianwei Yin, Xingyi Zhou, and Philipp Krahenbuhl. Center-based 3d object detection and tracking. In *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11784–11793, 2021.
- [11] Weijing Shi and Raj Rajkumar. Point-gnn: Graph neural network for 3d object detection in a point cloud. In *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1711–1719, 2020.
- [12] Pedro Felzenszwalb, David McAllester, and Deva Ramanan. A Discriminatively Trained, Multiscale, Deformable Part Model. In *Proc. of the IEEE Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.
- [13] Yuki Abe, Hiroshi Sasaki, Shinpei Kato, Koji Inoue, Masato Edahiro, and Martin Peres. Power and Performance Characterization and Modeling of GPU-accelerated Systems. In *2014 IEEE 28th international parallel and distributed processing symposium*, pages 113–122. IEEE, 2014.
- [14] D.G. Lowe. Object Recognition from Local Scale-invariant Features. *Proc. of the IEEE International Conference on Computer Vision*, 2:1150–1157, 1999.
- [15] P. Viola and M. Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. *Proc. of the IEEE Computer Vision and Pattern Recognition*, 1:511–518, 2001.
- [16] Navneet Dalal and Bill Triggs. Histograms of Oriented Gradients for Human Detection. In *Proc. of the IEEE Computer Vision and Pattern Recognition*, pages 886–893. IEEE, 2005.



- [17] P. F. Felzenszwalb, R. B. Girshick, and D. McAllester. Cascade Object Detection with Deformable Part Models. *Proc. of the IEEE Computer Vision and Pattern Recognition*, pages 2241–2248, 2010.
- [18] P. F. Felzenszwalb, R. B. Girshick, and D. McAllester. Object Detection with Discriminatively Trained Part-Based Models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.
- [19] J. Yan, Z. Lei, L. Wen, and S.Z. Li. The Fastest Deformable Part Model for Object Detection. *Proc. of the IEEE Computer Vision and Pattern Recognition*, pages 2497–2504, 2014.
- [20] M. Pedersoli, A. Vedaldi, and J Gonzàlez. A Coarse-to-fine approach for fast deformable object detection. *Proc. of the IEEE Computer Vision and Pattern Recognition*, pages 1353–1360, 2011.
- [21] H. Cho, P. E. Rybski, A. Bar-Hillel, and W. Zhang. Real-time Pedestrian Detection with Deformable Part Models. *Proc. of the IEEE Intelligent Vehicles Symposium*, pages 1035–1042, 2012.
- [22] T. Dean, M. A. Ruzon, M. Segal, J. Shlens, S. Vijayanarasimhan, and J. Yagnik. Fast, Accurate Detection of 100,000 Object Classes on a Single Machine. *Proc. of the IEEE Computer Vision and Pattern Recognition*, pages 1814–1821, 2013.
- [23] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.
- [24] The PASCAL Visual Object Classes Homepage. <http://pascallin.ecs.soton.ac.uk/challenges/VOC/>, 2014.
- [25] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755, 2014.

- [26] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361. IEEE, 2012.
- [27] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.
- [28] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. In *Proc. of the European Conference on Computer Vision*, 2016.
- [29] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual Explanations from Deep Networks via Gradient-based Localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.
- [30] Mariusz Bojarski, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Larry J Ackel, Urs Muller, Phil Yeres, and Karol Zieba. VisualBackProp: Efficient Visualization of CNNs for Autonomous Driving. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018.
- [31] Jinkyu Kim and John Canny. Interpretable Learning for Self-Driving Cars by Visualizing Causal Attention. In *Proceedings of the IEEE international conference on computer vision*, pages 2942–2950, 2017.
- [32] Markus Eisenbach, Ronny Stricker, Daniel Seichter, Alexander Vorn-dran, Tim Wengfeld, and Horst-Michael Gross. Speeding up deep neural networks on the jetson tx1. *Proc. of the Workshop on Computational Aspects of Pattern Recognition and Computer Vision with Neural Systems at international Joint Conference on Neural Networks*, 11, 2017.

- [33] Nils Tijtgat, Wiebe Van Ranst, Toon Goedeme, Bruno Volckaert, and Filip De Turck. Embedded real-time object detection for a uav warning system. In *Proc. of the IEEE International Conference on Computer Vision Workshops*, pages 2110–2118, 2017.
- [34] Riccardo Giubilato, Sebastiano Chiodini, Marco Pertile, and Stefano Debei. An evaluation of ROS-compatible stereo visual SLAM methods on a nVidia Jetson TX2. *Measurement*, 140:161–170, 2019.
- [35] R. B. Girshick, P. F. Felzenszwalb, and D. McAllester. Discriminatively Trained Deformable Part Models Release 5. <http://people.cs.uchicago.edu/~rbg/latent-release5/>.
- [36] P. F. Felzenszwalb and D. Huttenlocher. Distance Transforms of Sampled Functions. Technical report, Cornell Computing and Information Science Technical Report TR2004-1963, 2004.
- [37] Pedro F Felzenszwalb and Daniel P Huttenlocher. Pictorial structures for object recognition. *International journal of computer vision*, 61(1):55–79, 2005.
- [38] M. Hirabayashi, S. Kato, M. Edahiro, K. Takeda, T. Kawano, and S. Mita. GPU Implementations of Object Detection using HOG Features and Deformable Models. *Proc. of the IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, pages 766–771, 2013.
- [39] NVIDIA. Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture:Fermi. [http://www.nvidia.com/content/PDF/fermi\\_white\\_papers/NVIDIA\\_Fermi\\_Compute\\_Architecture\\_Whitepaper.pdf](http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf), 2009.
- [40] NVIDIA. Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture:Kepler GK110. <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>, 2012.

- [41] D. Stevenson et al. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, pages 9–25, 1987.
- [42] N. Whitehead and A. Fit-florea. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. <https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf>, 2011.
- [43] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *Proc. of the Advances in neural information processing systems*, pages 91–99, 2015.
- [44] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu. A Survey of Deep Learning-Based Object Detection. *IEEE Access*, 7:128837–128868, 2019.
- [45] Yusuke Mori and Kazuhiko Eguchi. Traffic Light Recognition from the Video Image of Home Video Camera. Technical report, Aichi institute of Technology, 2012.
- [46] Masako Omachi and Shinichiro Omachi. Traffic Light Detection with Color and Edge Information. In *Proc. of the IEEE International Conference on Computer Science and Information Technology*, pages 284–287, 2009.
- [47] Raoul de Charette and Fawzi Nashashibi. Traffic Light Recognition using Image Processing Compared to Learning Processes. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 333–338, 2009.
- [48] Mark Philip Philipsen, Morten Bornø Jensen, Andreas Møgelmoose, Thomas B Moeslund, and Mohan M Trivedi. Traffic light detection: A learning algorithm and evaluations on challenging dataset. In *Proc. of the IEEE conference on Intelligent Transportation Systems*, pages 2341–2345, 2015.

- [49] Chulhoon Jang, Sungjin Cho, Sangwoo Jeong, Jae Kyu Suhr, Ho Gi Jung, and Myoungho Sunwoo. Traffic light recognition exploiting map and localization at every stage. *Expert Systems With Applications*, 88:290–304, 2017.
- [50] Andreas Møgelmo, Dongran Liu, and Mohan M Trivedi. Traffic Sign Detection for U.S. Roads: Remaining Challenges and a case for Tracking. In *Proc. of the IEEE conference on Intelligent Transportation Systems*, pages 1394–1399, 2014.
- [51] Yongtao Yu, Jonathan Li, Chenglu Wen, Haiyan Guan, Huan Luo, and Cheng Wang. Bag-of-visual-phrases and hierarchical deep models for traffic sign detection and recognition in mobile laser scanning data. *ISPRS Journal of Photogrammetry and Remote Sensing*, 113:106–123, 2016.
- [52] HERE Global B.V. HERE HD Live Map. <https://here.com/en/file/8596/download?token=EFFgFGKC>, 2016.
- [53] Peter Biber and Wolfgang Straßer. The normal distributions transform: A new approach to laser scan matching. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2743–2748, 2003.
- [54] Eijiro Takeuchi and Takashi Tsubouchi. A 3-D scan matching using improved 3-D normal distributions transform for mobile robotic mapping. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3068–3073, 2006.
- [55] Mengwen He, Huijing Zhao, Jinshi Cui, and Hongbin Zha. Calibration method for multiple 2d LIDARs system. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 3034–3041, 2014.
- [56] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Hamada Tsuyoshi. An Open Approach to Autonomous Vehicles. *IEEE Micro*, 35(6):60–68, 2015.

- [57] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.
- [58] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *Workshop on Open Source Software of the IEEE International Conference on Robotics and Automation*, volume 3, page 5, 2009.
- [59] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proc. of the ACM international conference on Multimedia*, pages 675–678, 2014.
- [60] Ross Girshick. Fast R-CNN. In *Proc. of the IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.
- [61] Igor Orlov. Trafficlight. <https://github.com/igororlov/TrafficLight>.
- [62] Manato Hirabayashi, Shinpei Kato, Masato Edahiro, Kazuya Takeda, and Seiichi Mita. Accelerated Deformable Models on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1589–1602, 2016.
- [63] Nathaniel Fairfield and Chris Urmson. Traffic Light Mapping and Detection. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 5421–5426, 2011.
- [64] Gaurav Pandey, James R McBride, Silvio Savarese, and Ryan M Eustice. Automatic Targetless Extrinsic Calibration of a 3D Lidar and Camera by Maximizing Mutual Information. In *Proc. of the AAAI Conference on Artificial Intelligence*, pages 1293–1300, 2012.

- [65] Andreas Geiger, Frank Moosmann, Ömer Car, and Bernhard Schuster. Automatic Camera and Range Sensor Calibration using a single Shot. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 3936–3943, 2012.
- [66] Davide Scaramuzza, Ahad Harati, and Roland Siegwart. Extrinsic Self Calibration of a Camera and a 3D Laser Range Finder from Natural Scenes. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4164–4169, 2007.
- [67] S. Kato, J. Aumiller, and S. Brandt. Zero-Copy I/O Processing for Low-Latency GPU Computing. *Proc. of the IEEE/ACM International Conference on Cyber-Physical Systems*, pages 170–178, 2013.
- [68] A. Nguyen, Y. Fujii, Y. Iida, T. Azumi, N. Nishio, and S. Kato. Reducing Data Copies between GPUs and NICs. *Proc. of the IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, pages 37–42, 2014.
- [69] Y. Iida, M. Hirabayashi, T. Azumi, N. Nishio, and S. Kato. Connected Smartphones and High-Performance Servers for Remote Object Detection. *Proc. of the IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, pages 71–76, 2014.
- [70] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. *Proc. of the USENIX Annual Technical Conference*, pages 1–14, 2011.
- [71] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-Class GPU Resource Management in the Operating System. *Proc. of the USENIX Annual Technical Conference*, pages 1–12, 2012.
- [72] C. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. *Proc. of the ACM Symposium on Operating Systems Principles*, pages 233–248, 2011.

- [73] NVIDIA. Jetson TX1. <https://developer.nvidia.com/embedded/jetson-tx1>.
- [74] NVIDIA. Jetson TX2. <https://developer.nvidia.com/embedded/jetson-tx2>.
- [75] NVIDIA. Jetson AGX Xavier. <https://developer.nvidia.com/embedded/jetson-agx-xavier>.
- [76] NVIDIA. Jetson Nano. <https://developer.nvidia.com/embedded/jetson-nano>.
- [77] arm. Mali series. <https://www.arm.com/en/products/silicon-ip-multimedia>.
- [78] Sparsh Mittal. A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *Journal of Systems Architecture*, 2019.
- [79] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.
- [80] NVIDIA. TensorRT. <https://developer.nvidia.com/tensorrt>.
- [81] The Autoware Foundation. Autoware. <https://gitlab.com/autowarefoundation/autoware.ai>.
- [82] COCO Consortium. COCO 2017 Val images. <http://images.cocodataset.org/zips/val2017.zip>.
- [83] NVIDIA. CUDA C++ BEST PRACTICES GUIDE. [https://docs.nvidia.com/cuda/pdf/CUDA\\_C\\_Best\\_Practices\\_Guide.pdf](https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf), 2019.
- [84] COCO Consortium. COCO 2014 Val images. <http://images.cocodataset.org/zips/val2014.zip>.



- [85] Senthil Yogamani, Ciarán Hughes, Jonathan Horgan, Ganesh Sistu, Padraig Varley, Derek O’Dea, Michal Uricár, Stefan Milz, Martin Simon, Karl Amende, et al. WoodScape: A multi-task, multi-camera fisheye dataset for autonomous driving. *arXiv preprint arXiv:1905.01489*, 2019.

# List of Publications by the Author

## Research achievements related to the dissertation

### Journal papers

1. Manato Hirabayashi, Shinpei Kato, Masato Edahiro, Kazuya Takeda, and Seiichi Mita, “Accelerated Deformable Part Models on GPUs”, *IEEE Transactions on Parallel & Distributed Systems*, Vol. 27, No. 6, pp. 1589–1602, Jun. 2016.
2. Manato Hirabayashi, Adi Sujiwo, Abraham Monrroy, Shinpei Kato, and Masato Edahiro, “Traffic Light Recognition using High-Definition Map Features”, *Journal of Robotics and Autonomous Systems*, Vol. 111, pp. 62–72, Jan. 2019.
3. Manato Hirabayashi, Yukihiro Saito, Kosuke Murakami, Akihito Ohsato, Shinpei Kato, and Masato Edahiro , “Vision-based Sensing Systems for Autonomous Driving: Centralized or Decentralized?”, *Journal of Robotics and Mechatronics*, Vol. 33, No. 3, pp. 686–697, Jun. 2021

### International conferences

1. Manato Hirabayashi, Shinpei Kato, Masato Edahiro, Kazuya Takeda, Taiki Kawano, and Seiichi Mita, “GPU Implementations of Object

Detection using HOG Features and Deformable Models”, in Proc. of the IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications, pp. 106–111, 2013.

## **Other research achievements**

### **International conferences**

1. Yuki Iida, Manato Hirabayashi, Takuya Azumi, Nobuhiko Nishio, and Shinpei Kato, “Connected Smartphones and High-Performance Servers for Remote Object Detection”, in Proc. of the IEEE 2nd International Conference on Cyber-Physical Systems, Networks, and Applications, pp. 71–76, 2014.
2. Yuki Kitsukawa, Manato Hirabayashi, Shinpei Kato, and Masato Edahiro, “Exploring the Problem of GPU Programming for Data-Intensive Applications: A Case of Multiple Expectation Maximization for Motif Elicitation”, in Proc. of the 5th Symposium on Information and Communication Technology, pp. 256–262, 2014.
3. Abraham Monrroy, Manato Hirabayashi, Shinpei Kato, Masato Edahiro, Takefumi Miyoshi, and Satoshi Funada, “Hexa Cam: An FPGA-Based Multi-view Camera System”, in Proc. of the IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications, pp. 48–53, 2015.

### **International workshop**

1. Manato Hirabayashi, Shinpei Kato, Masato Edahiro, and Yuki Sugiyama, “Toward GPU-Accelerated Traffic Simulation and Its Real-Time Challenge”, in Proceedings of the 1st International Workshop on Real-Time and Distributed Computing in Emerging Applications, pp. 1–6, 2012.