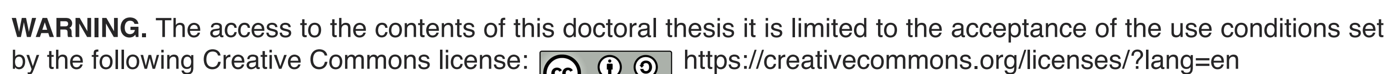
Universitat Autònoma de Barcelona

ADVERTIMENT. L'accés als continguts d'aquesta tesi queda condicionat a l'acceptació de les condicions d'ús establertes per la següent llicència Creative Commons: http://cat.creativecommons.org/?page_id=184

ADVERTENCIA. El acceso a los contenidos de esta tesis queda condicionado a la aceptación de las condiciones de uso establecidas por la siguiente licencia Creative Commons: http://es.creativecommons.org/blog/licencias/

WARNING. The access to the contents of this doctoral thesis it is limited to the acceptance of the use conditions set by the following Creative Commons license:  https://creativecommons.org/licenses/?lang=en

Reconfigurable HOG/SVM Implementations for Pedestrian Detection

QUANG-VINH NGO

Thesis Advisors:
Professor **Jordi Carrabina-Bordoll**
Dr. **David Castells-Rufas**

Ph.D. program:
Electronic and Telecommunication Engineering

Department: Microelectronics and Electronic Systems, School of Engineering

This dissertation is submitted for the degree of

Doctor of Philosophy

April 2022

Declaration

I hereby declare that, except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University.

QUANG-VINH NGO
April 2022

Acknowledgements

First and foremost, I would like to express my deepest appreciation to my directors, Professor Jordi Carrabina and Dr. David Castells-Rufas. You not only taught me the scientific way of thinking and working but also an industrial approach. In addition, your encouragement and patience have given me the positive energy to fulfill both projects and thesis.

The research is sponsored by a Personal Investigador en Formación (PIF) Ph.D. grant from the Universitat Autònoma de Barcelona (420-2017). The completion of my Ph.D. program would not have been possible without this sponsor.

I would also like to extend my sincere thanks to my colleagues in UAB: Marc Codina, Arnau Casadevall, and Juan Borrego. Your collaboration has been consistently professional and productive.

To my beloved friends in Spain, Marco, Jesus, Jordi Caballero, and Simon, you have always been there for me, both to talk and teach me social skills. Thanks to you, I was able to establish work-life balance and improve my performance.

Finally, I am extremely grateful to my family, my wife, daughter, and son. Dad, you have been waiting for this for several years. Thank you for your endless love.

Abstract

Pedestrian detection is one of the most safety-critical applications in autonomous cars. The requirement of this application is not only accuracy but also speed and energy efficiency. In the literature, there are two main approaches to solve the problem: deep neural network based algorithms, that achieve high accuracy yet require large amount of computing resource and power; and handcrafted features based classifications, more suitable for embedded platforms with limited amount of computing and memory resources.

Embedded platforms implemented using FPGAs and ASICs consume less power than GPU/CPU based systems to achieve similar results. On the other side, in terms of energy efficiency, GPUs is 10 times better than FPGAs in running CNN-based applications. However, FPGA-based implementations with low-level optimization techniques can beat GPU-based ones. Compared to ASICs, the advantage of FPGA device is that it is their re-configurability for later updates and time-to-market.

This thesis presents the implementation of pedestrian detection systems on FPGAs using Histogram of Gradient feature extractor and SVM classifier. First, the pipeline of the algorithm is implemented in Verilog HDL to achieve a high-throughput and low power consumption system. Second, the same algorithm is realized using OpenCL programming model, a high-level synthesis approach. To compare to the state-of-the-art, since different implementations have different working frequencies and input image resolution, I calculate the number of pixels per clock cycle for fair comparison. The implementation in this thesis achieves second to the best with 0,068 pixels per clock even though it uses equal or less FPGA resources than the rest. The system consumes the least power at only 9 W. In terms of energy efficiency, our result achieves the third best at 1,22 FPS per Watt. However, the working frequency of this design is only half as high as the frequencies of the other implementations. If the pixel clock is doubled to be 100 MHz, the energy efficiency of this design would become the best.

Table of contents

List of figures **xv**

List of tables **xix**

1 Motivation **1**

1.1 Problem statement 3

1.2 Hypothesis 3

1.3 Objective 4

1.4 Methodology 4

1.5 Structure of the dissertation 5

2 FPGA technology for autonomous cars **7**

2.1 Overview of autonomous cars 7

2.2 History of autonomous cars 11

2.3 Vision-based applications on autonomous cars 14

2.3.1 Image sensors 14

2.3.2 Evaluation metrics 16

2.3.3 Applications 18

2.3.4 Depth sensing 19

2.3.5 Semantic segmentation 20

2.3.6 Object detection 23

2.3.7 Lane detection 25

2.3.8 Traffic signs recognition 27

2.3.9 Traffic lights recognition 27

2.3.10 Obstacle detection 28

2.4 Hardware platforms for autonomous cars 31

2.5 FPGA technology 31

2.5.1 Design methodologies 32

2.6 FPGAs for vision-based algorithms . . . 33
2.6.1 FPGA common techniques . . . 38

3 Pedestrian detection on autonomous cars 41
3.1 Pedestrian detection . . . 41
3.2 Evaluation methodology . . . 45
3.3 State of the Art . . . 49
3.4 HOG/SVM Pedestrian Detection . . . 52
3.4.1 Gamma normalization . . . 53
3.4.2 Gradient computation . . . 53
3.4.3 Orientation bin voting . . . 54
3.4.4 Block Normalization . . . 55
3.4.5 Support Vector Machine . . . 56
3.5 Summary . . . 60

4 HOG/SVM pedestrian detection implementation 61
4.1 System architecture . . . 61
4.2 HOG extractor pipeline design . . . 64
4.2.1 The pipeline design . . . 64
4.3 SVM Classifier pipeline design . . . 73
4.4 Results . . . 78
4.4.1 Accuracy . . . 78
4.4.2 Latency . . . 79
4.4.3 Throughput . . . 80
4.4.4 Comparison with existing implementations . . . 80
4.5 Summary . . . 83

5 HOG/SVM pedestrian detection implementation using OpenCL 85
5.1 OpenCL programming model . . . 85
5.1.1 Optimization techniques . . . 88
5.2 OpenCL implementation . . . 92
5.3 Results and comparison . . . 105
5.3.1 Comparison with multicore CPU . . . 105
5.3.2 Comparison between FPGA platforms . . . 106
5.3.3 FPGA-based pedestrian detection comparison . . . 109
5.3.4 Comparison with other OpenCL-based design . . . 110
5.4 Summary . . . 110

6 Conclusions and Future Directions 113
6.1 Conclusions 113
6.2 Future directions 114

References 115

Appendix A Calculate the FPS of output images from the sensor 129
A.1 Camera settings 129
A.2 Frame rate calculation 130

Acronyms

ACC Adaptive Cruise Control. 11

ADAS Advanced Driver Assistance Systems. 1

API Application Programming Interface. 88

CV Computer vision. 19

DARPA Defense Advanced Research Projects Agency. 11

DNN Deep Neural Network. 23

DSP Digital Singal Processing. 109

FPGA Field Programmable Gate Array. 4

FPPI False Positive Per Image. 46, 78

GPU Graphics processing unit. 2

HDL Hardware Description Language. 109

II Initiation interval. 90

IoU Intersection over Union. 16, 17

LiDAR Light Detection and Ranging. 1

NMS Non-Maximum Suppression. 46

PCIe Peripheral Component Interconnect Express. 33, 85

PDS pedestrian detection system. 3

QPI QuickPath Interconnect. 33

SDRAM Synchoronous Dynamic Random Access Memory. 88

SoA State of the art. 27

TFLOPs Tera Floating-point Operations. 1

TLR Traffic Lights Recognition. 23, 27

TSR Traffic Signs Recognition. 23, 27

List of figures

2.1 Typical sensors on an autonomous car 7
2.2 Typical functions of a self-driving car 8
2.3 Five levels of autonomy 10
2.4 Example of the Intersection of a detected area with a ground-truth recording (left) and the area of their union (right). 17
2.5 Precision Recall curve example 17
2.6 Typical vision-based algorithms of a self-driving car 18
2.7 Example of the disparity map computed using SGM from stereo images provided in [23] 21
2.8 Semantic segmentation example Cityscapes dataset [26]. Objects in the same class have the same annotation color. 22
2.9 Instance segmentation example from the Synthia synthetic dataset [27]. Each person has a unique annotation color. 23
2.10 Detection result example on COCO dataset done in [40]. The ground truth objects at pixel granularity and the predictions are bounding boxes 24
2.11 Typical processing steps of a CNN-based object detection algorithm. 25
2.12 Fragment of a challenging scenario for lane detection with a corresponding pixel level ground-truth for visible lane marks (center), and partially ocluded road lanes (right). 26
2.13 Multiple signs on both sides at the far sight including information, warning, and prohibitory signs. b) Two attaching prohibitory signs and an information sign with a secondary sign attached. 28
2.14 Traffic lights detection example in Germany [67]. 29
2.15 Example of obstacles on a road. 30
2.16 Main replicated FPGA components interconnected by a configurable interconnect. 32
2.17 Typical structures of an FPGA-based accelerator 34

2.18 CV generic pipeline 35
2.19 Neural network accelerator platforms [6]. 36
2.20 An illustration of line buffer. 39
2.21 Example of task parallelism at the fine-grain (left) and a coarse grain (right) levels 39

3.1 Example images from Caltech Dataset with annotations. 42
3.2 Calculation the distance from the height of a pedestrian 47
3.3 Sliding a 7x15 window over a 79x59 image 49
3.4 Block diagram of a conventional pedestrian detection system. 49
3.5 Top detectors on Caltech Pedestrian detection benchmark [116] 50
3.6 Block diagram of a HOG+SVM pedestrian detection algorithm 52
3.7 HOG feature extractor block diagram 52
3.8 An illustration of how HOG features are generated. a) Image is divided in 8x8 cells. The pixels' value of a cell are randomly created to plot the histogram in the subfigure 3.8c. The gradient vector is also illustrated; each cell has $8x8$ pixels. b) A pixel has a magnitude gradient G, and orientation gradient ϕ ranged from 0 to 180°; these gradients are calculated from G_x and G_y. c) All 64 pixels' values in subfigure 3.8a vote their magnitude gradient to the appropriate orientation bin among 9 bins to create the cell 's vector. . 55
3.9 Illustration of cell, block, and detection window in an image. The detection window will slide all over the image to detect object of interest. Each window has a size of 7x15 blocks, each block is 2x2 cell, and each cell is 8x8 pixel. 56
3.10 Illustration of 1) Circled data points which are also called support vectors, 2) The bold line $f(\vec{x})$ which separates the two types of data points: positive symbol and negative symbol, 3) Data points in rectangular shape which are on the wrong side of the bold line, 4) Data points belong to two classes, represented by positive symbols and negative symbols, are separated correctly, 5) The margin corresponding to the bold line that separates data points. SVM selects the bold line so that the margin is biggest while the number of rectangular points is the smallest. 58

4.1 System diagram of the RTL-based pedestrian detection system on DE1-SoC. 62
4.2 Bayer pattern illustration. 63
4.3 HOG extractor block diagram 64
4.4 Pixel line buffers 65
4.5 CORDIC IP functionality settings 67

4.6 CORDIC IP performance setting 68
4.7 Illustration of an 8*x*8 cell in a 640*x*480 image 68
4.8 AGGREGATE module block diagram 69
4.9 Blocks and cells in an 640x480 image 70
4.10 Normalization finite state machine 72
4.11 Normalization pipeline 73
4.12 Sliding-window and convolution illustration. 74
4.13 Illustration of a block being part of different windows 76
4.14 SVM classifier hardware block diagram 77
4.15 Comparison of our model with the standard HOG on Caltech pedestrian dataset 79
4.16 Latency of the whole pipeline 80

5.1 OpenCL programming model 86
5.2 OpenCL memory model 87
5.3 Pipelined loop latency: L iterations 90
5.4 Hardware-software inter-operation based on OpenCL 93
5.5 Latency comparison between modules on different FPGAs 109

List of tables

2.1 Stereo depth estimation datasets 20
2.2 Semantic segmentation datasets 23
2.3 Lane detection datasets 26
2.4 Best traffic sign recognition implementations on GTSDB and STSD 27
2.5 Traffic light recognition datasets and their best implementations 29
2.6 Automakers and their computing hardware platforms 31
2.7 Comparison of different hardware platforms for autonomous cars 36
2.8 Energy efficiency of algorithms on FPGAs and GPUs 37

3.1 Popular pedestrian datasets 44
3.2 Scaled image sizes 48
3.3 Performance results of detectors on Caltech dataset (the information on the Table is extracted from Caltech pedestrian dataset website [116]). 51
3.4 Some popular datasets and the best implementations on them. 52
3.5 Different masks for gradient computation 53

4.1 Image resolution from the camera to the HOG Extractor 63
4.2 Comparison with the state of the art 82

5.1 Gradient kernel's resource and performance on different FPGAs 104
5.2 Resources usage and performance of the Histogram kernel. The first is the version with fixed-point and line buffer. The second version removes the line buffer. The other versions use float data type. The last version in the Table implements shift registers to reduce loop-carried data dependency. 104
5.3 Histogram kernel's resource and performance on different FPGAs 104
5.4 Resources usage and performance of the Normalization kernels. One version is L1 normalization with nested loops. The other version is L1-sqrt without nested loops. 104
5.5 Normalization kernel's resource and performance on different FPGAs . . . 105

5.6 Resources usage and performance of the SVM classifier kernels: version with no shift registers and with shift registers 105
5.7 SVM classifier kernel's resource and performance on different FPGAs . . . 105
5.8 Latency comparison with multicore CPU 107
5.9 Resources usage and performance on different FPGA platforms 107
5.10 Throughput comparison with other heterogeneous platforms 108
5.11 Lines of code comparison between HDL-based and OpenCL-based approach 110

A.1 Pixel clock settings 129
A.2 Frame rate calculation 131

Listings

3.1 NMS pseudo-code 45
5.1 Example of constant keyword [1] 88
5.2 Example of restrict keyword [1] 89
5.3 Implementation of a line buffer 89
5.4 Floating-point adder without shift registers 91
5.5 Floating-point adder with shift registers 91
5.6 OpenCL code for the Gradient kernel 94
5.7 OpenCL code for the Histogram kernel 96
5.8 OpenCL code for the Normalization kernel 101
5.9 OpenCL code for the SVM kernel 102

Chapter 1

Motivation

Safety is of paramount important for road users, whether they be drivers (cars, trucks, buses, trainways), or bikers, cyclists, and pedestrians. Most road accidents are linked to drivers. Governments have therefore strengthened legislation - such as speed limit, alcohol testing, etc. - to mitigate key risk factors; however, the number of fatalities due to road traffic crashes continues to climb. In 2016, this number was estimated at 1.35 million worldwide, which is nearly 3,700 mortalities each day [2].

One of the solutions to this problem comes from Advanced Driver Assistance Systems (ADAS) technology, the aim of which is to help reduce human errors while driving. The technology senses the surrounding environment using sensors like cameras, sonars, radars, LiDARs, etc. Advanced Driving Assistance Systems (ADAS) consists of a computing system which generates a perception model from the sensors' data, analyzes the situation, and thus provides appropriate decisions. These decisions are then realized on the actuators of the car. The safety-critical ones are the throttle pedal, the braking pedal, the gearbox, and the steering wheel. However, the ADAS system also controls lights, screen wipers, and other functions of the car.

The computing system in ADAS needs to perform any required task in real-time with low energy consumption. However, most state-of-the-art algorithms for ADAS are based on machine learning algorithms that are computationally complex to implement. For instance, an object detection system based on Yolo3 neural network already requires a processing capability of 1.457 TFLOPs per second to achieve a throughput of 78 FPS on a limited 256x256 resolution [3]. This number of operations per second is very high, especially when several algorithms must be executed concurrently for the different ADAS tasks and sensors in the same computing system. The ADAS includes but is not limited to traffic light recognition, traffic sign recognition, lane detection, and pedestrian detection [4]. Besides the processing speed requirement, the computing system needs to consume low levels of energy since energy

efficiency is crucial to achieve longer autonomy which is one of the most important factors considered by users when selecting between a fuel car or an electric car. To achieve those requirements, the algorithms need to be implemented on an appropriate hardware platform.

There are different hardware platforms which are used to implement such computing systems such as: (i) ASICs, (ii) FPGAs; (iii) CPUs, (iv) GPUs; or (v) combinations of them in homogeneous or heterogeneous multi-processing platforms. However, each of them provides a different balance between throughput and energy efficiency.

In the hardware implementations side, ASICs are the most energy efficient implementations, but they are not as flexible as FPGAs, since their HW architecture cannot be updated. By contrast, when using FPGAs, new hardware features can be reprogrammed with flexibly – such as those related to new sensors or actuators. Furthermore, ASICs usually have higher Non-Recurring Engineering (NRE) costs, especially when using similar technology nodes.

Comparing to software implementations in CPUs and GPUs, FPGAs are known to be more energy efficient in some applications [5]. In the case of neural network inference accelerators, FPGAs can be at least 10 times better than GPUs as a main implementation competitor [6]. One reason is that FPGA-based implementations are strongly tailored for a specific application, while software (SW) platforms usually have software and hardware designed independently. Thus, designers have more flexibility to optimize FPGA implementations of specific applications, for speed and energy efficiency. For instance, the working-frequency on FPGAs can be selected such that it is, at the same time, sufficient to achieve required speed for real-time conditions while minimal to save energy consumption.

In terms of latency, FPGAs also have advantages over CPUs and GPUs. One of the most distinct capabilities of an FPGA is that it can to a great extent parallelize computational tasks, from hardware parallelization (spatial concurrency) to hardware pipelining (temporal concurrency) and their combinations. Moreover, FPGA devices can be connected directly to other sensors or devices, which helps reducing the total latency compared to that of CPUs and GPUs.

Additionally, the latency of FPGA-based designs can be made deterministic [7], [8], which is often a requirement for the car control system, especially for safety-critical functions. In [8], researchers evaluate image acquisition and processing system for automated driving systems. The paper shows that latency deviation with a host CPU in the input data path is up to 77.4 milliseconds; however, without the host CPU, the latency deviation of the FPGA-only design is reduced to 115.5 nanoseconds.

Therefore, when efficiency, latency, and flexibility are taken into consideration, FPGAs represent the most suitable hardware option to compensate their usually higher costs.

1.1 Problem statement

This research addresses the pedestrian detection system (PDS) for ADAS, one of the most safety-critical applications of autonomous cars. This is a sub-problem of obstacle detection; however, because of its ethical and legal importance, it is usually considered separately.

The challenge for this technology is to detect both stationary and moving people in the area in front of the car (pedestrians are mostly found here in car-to-pedestrian accidents) [9]. Detecting the presence and the position of pedestrians helps to warn the driver, brake, and/or deploy external airbags.

However, despite the high stakes, the key challenges for a pedestrian detection system on autonomous cars are like other ADAS algorithms: real-time processing speed, low power-consumption, and high detection accuracy. Besides, pedestrians come in many forms – in different poses, wearing different clothes, possessing different heights. This high variability challenges the detection algorithm. In some cases, pedestrians can be occluded. The outdoor environment poses another difficulty for the algorithm, since variations in light and weather conditions can affect both the sensor and algorithmic performance.

1.2 Hypothesis

In the literature, pedestrian detection systems are typically based on vision processing algorithms. Machine learning techniques such as deep neural networks provide very high accuracy but also demand high computing resources and power consumption. Classical machine learning methods (hand-crafted features based), such as Histogram of gradients (HOG) plus Support Vector Machines (SVM), are less computing-intensive and more suitable for embedded platforms such as FPGAs. HOG is the most accurate feature for pedestrian detection and SVM as their classifier.

The hypothesis which guides this thesis is that the implementation of pedestrian detection systems on FPGAs using HOG and SVM can achieve high processing speed while being energy efficient with an acceptable detection accuracy. The hypothesis can be posed in the following research questions:

- What trade-off can be achieved by a HOG PDS between processing speed and detection accuracy in FPGA platforms?

- Are FPGAs more energy efficient than other implementations on different platforms for an equivalent processing speed and detection accuracy?

1.3 Objective

The thesis targets the implementation of a high performance and energy efficient PDS for ADAS, based on FPGA technology. The objective is to tailor the parameters and architecture of the hardware implementation of pedestrian detection algorithms in real-time on FPGAs to obtain an optimal trade-off of accuracy, speed, area, and energy efficiency.

I will use HOG as the feature detection and SVM to classify pedestrians due to its demonstrated validity at algorithmic level and popularity. The system should detect pedestrians at least at a real-time throughput of 30 FPS (@640x480) as commonly accepted as real-time image processing speed. The accuracy of the system is evaluated using the Caltech dataset, which is considered suitable for autonomous cars since its image data derive from real street views.

In order to find the near optimal implementations, architectural space exploration will be addressed using an OpenCL-based framework to quickly generate and test a customizable PDS deployed on FPGAs.

1.4 Methodology

Many modern FPGAs system-on-chips (SoCs) combine an ARM processor with some FPGA fabric. HOG and SVM are both in the critical path of the pedestrian detection system and therefore the architecture and detailed implementation of these two modules need to be optimized on the FPGA's logic part while software-friendly functions such as labelling or drawing detection boxes can be implemented on FPGA's embedded processor (e.g. ARM).

The research includes two stages of development. At first, HOG and SVM are described at RTL level to describe the algorithm in HW. RTL is often used in creating FPGA designs. It can describe the microarchitecture of the design at the lowest abstraction level, as a set of registers, memories, and combinational logic. RTL flow allows hardware engineers to optimize the implementation down to the finest level of granularity. However, it usually requires a long period of coding and testing. In this research, to verify the correctness of every hardware block, a reference model is written in C/C++, which has the same functional blocks as in the hardware RTL.

In the second stage, the pedestrian detection system is described in OpenCL to explore different configurations. OpenCL (Open Computing Language) is a framework for writing code that execute across heterogeneous platforms including FPGAs. OpenCL language is similar to C and it can be written and tested within a shorter time than RTL code. The code under test can be emulated in CPUs for testing and debugging. Therefore, designers can

explore architectural space efficiently. OpenCL-based implementations are scalable and flexible, and allow accelerating different parts of the code using different technologies such as GPUs and FPGAs.

1.5 Structure of the dissertation

Chapter 2 provides an overview of ADAS and how FPGA technology can be used in the system. Chapter 3 provides an in-depth exploration of the pedestrian detection problem and the state-of-the-art technology for addressing it. In chapter 4, the HDL design of the system is described in detail, implemented and tested and then compared with the state-of-the-art implementations. Chapter 5 addresses the architectural space exploration of the pedestrian detection system using the OpenCL approach. Achieved performance is also discussed and compared with the RTL design. Finally, Chapter 6 gives the conclusion.

Chapter 2

FPGA technology for autonomous cars

2.1 Overview of autonomous cars

Autonomous cars are referred to by different names: self-driving car, robotic car, robo-car, driverless car, or autonomous vehicle. Any of these names imply that the vehicle employs electronic technology to sense the surrounding environment and drive safely, with -depending on the degree of autonomy - little to no human input. From a technology perspective, an autonomous system needs to cover three main tasks: perceiving, processing, and actuating. The corresponding hardware components are the sensors, the computing system, and the actuators. The most common sensors are image sensors, radar, lidar, ultrasonic, GNSS, and inertial measurement units. Figure 2.1 illustrates these sensors on an autonomous car.

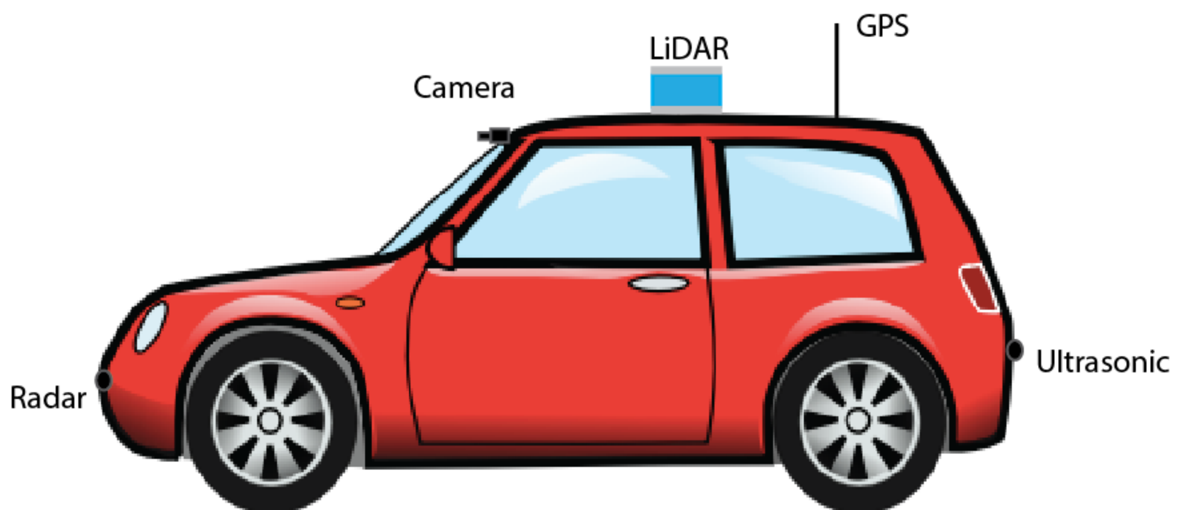


Fig. 2.1 Typical sensors on an autonomous car

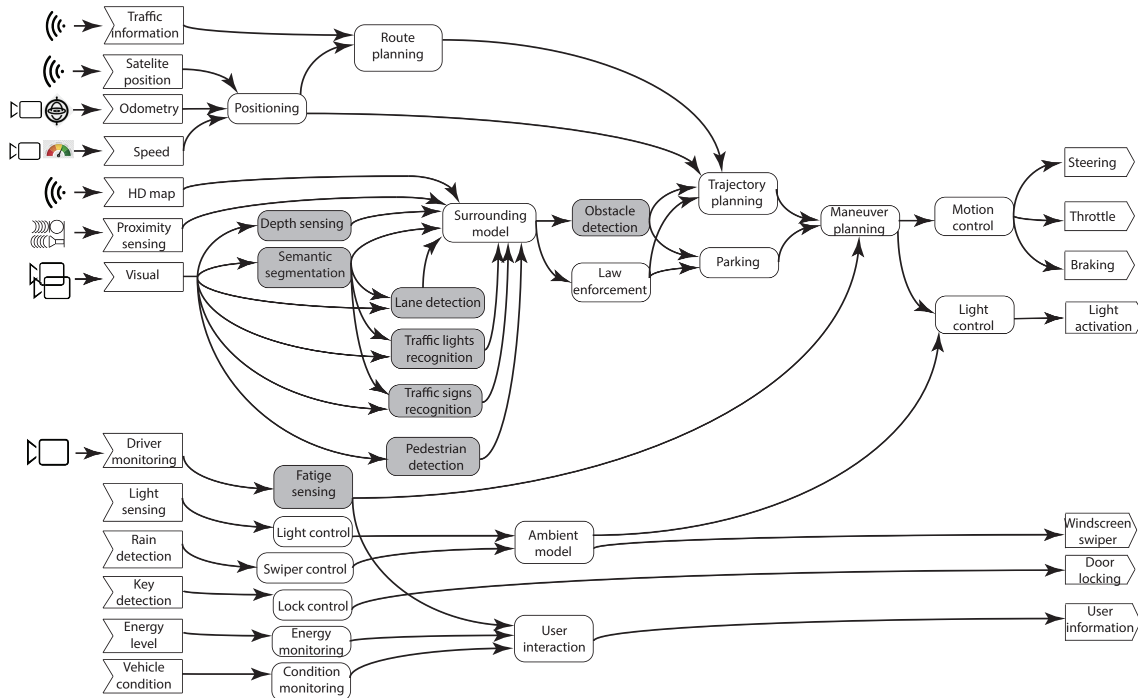


Fig. 2.2 Typical functions of a self-driving car

Based on data received from the sensors, processing algorithms in the central computing system analyze the scene to provide decisions. The decision feeds through to actuators to safely navigate the car. The typical algorithms of an autonomous car are presented in Fig. 2.2 as rounded rectangles. The shaded boxes are the vision-based ones. The input to the algorithms comes from sensors (on the left side of the figure). The right side illustrates actuators taking the output from the algorithms.

In Fig. 2.2, the top half describes safety-critical functions which control the steering, the throttle, the brake, and the car's lights. The motion control module manipulates the steering wheel, the throttle, and the brake pedals. It operates with instructions from the maneuver planning module. This module defines the final and specific path that the car will traverse. The path can be determined by the parking module or the trajectory planning module depending on the context. The output of these two modules should contain concrete instructions on how to move. They both get the input data from the two modules: (1) obstacle detection and (2) law enforcement, described as follows:

- Obstacle detection: in this context, this could be pedestrians, vehicles, barriers, walls or other objects. Obstacles are detected by analyzing the surrounding environment provided by the surrounding model module that is built upon the fusing data from high-

definition (HD) map and proximity sensors. Additionally, the surrounding model takes outputs information from the depth sensing module, semantic segmentation module, and the lane, traffic signs, traffic lights, pedestrian detection modules. Depending on the specific approach for detection, lane, traffic signs, and traffic lights detection modules can either take the visual data only as input or fuse with the segmentation information from the semantic segmentation module.

- Law enforcement: in addition to obstacle avoidance, traffic law must be calculated to determine the most appropriate path and the suitable speed. The law enforcement module obtains data from the output of the surrounding model module.

The trajectory planning module also requires information from the route planning and position module. The route planning module outputs possible routes, based on the current position and traffic information, to the trajectory planning module. The current position can be calculated from the GNSS (Global Navigation Satellite System), odometry, and speed information.

The bottom half of Figure 2.2 describes other autonomous functions which are considered less safety critical. The ambient model module perceives the outside environment regarding light intensity and other special weather conditions such as rain, fog, or others. The light control module will activate the appropriate car light for each brightness or weather condition. This module also controls the turning lights or stop light depending on the output of the Maneuver planning module. Similarly, the wiper control module uses rain sensor information to activate the windscreen wiper. The lock control module will lock the doors when it detects a certain condition for safety. Finally, the user interaction module can warn the driver regarding energy level and vehicle condition, by graphic displaying or even alarming. Fatigue sensing also helps to detect the drowsiness of a driver, based on a camera in front of him/her mounted inside the car.

Having established these basics, it is now necessary to address the different levels of autonomy standardized. To support the legislation of autonomous cars, the US Department of Transportation classified them into 5 levels of autonomy [10]. Figure 2.3 details the key difference between levels.

The lowest level of autonomy is Level 1, in which the driver is assisted by functions such as a braking or steering system, one function at a time. The vehicle can achieve lane centering or adaptive cruise control (ACC). At this level, the driver needs to fully control the vehicle. Level 2, named Partial Automation, also requires the driver to be in place and to monitor the environment at all times. The car simultaneously supports steering wheel and brake control. At this level upward, vehicles sense the surrounding environment. Advance

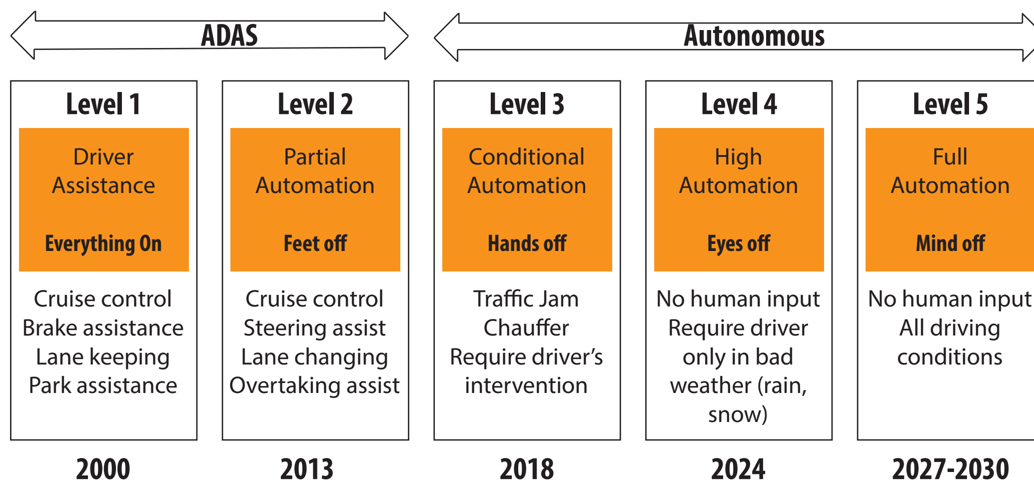


Fig. 2.3 Five levels of autonomy

features in Level 2 (also known as Level 2+) include lane changing, lane merging, and highway exiting and entering. Typical Level 2 systems in industry are General Motors Super Cruise, Mercedes-Benz Drive Plot, Tesla Autopilot, Volvo Pilot Assist, and Nissan ProPilot Assist 2.0. Specifically, Mercedes S Class 2014 and Tesla Autopilot 2014 are autonomous (with the driver always monitoring the system) in steering, lane keeping, acceleration, and braking on the high-way [11]. At Level 3, a driver is needed but is not required to monitor the environment under certain conditions; the car can pass other slow-moving cars and drive in traffic jams and specific highways. The driver, however, must always be ready to take control of the vehicle. For this reason, some car manufacturers, such as Ford, Google, and Volvo, skip this level and proceed with Level 4. In 2021, Honda became the first carmaker to sell Level 3 car with their Honda Sensing Elite.

Level 4 pertains to high-level automation in which the vehicle can perform all driving functions under certain conditions or areas. The key difference from Level 3 is that the cars at this level can manage when something goes wrong without the intervention of the driver. Due to current legislation and infrastructure, Level 4 cars can only work in limited areas and in certain weather conditions. However, companies are currently targeting Level 4 cars for driverless taxis or public transportation services. Such services have specific geographic boundaries which cars learn in advance. Waymo, originated as a project of Google, has been testing their Level 4 driverless cars in Arizona, US, since 2017.

Finally, Level 5 vehicles can perform all functions under all conditions, meaning full automation. This means that the vehicle can perform multiple tasks such as adaptive cruise control, traffic sign recognition, traffic light recognition, lane departure warning, emergency

braking, pedestrian detection, collision avoidance, cross traffic alert, surround view, park assist, rear collision warning or park assistance.

2.2 History of autonomous cars

The history of autonomous cars is comprehensively represented in [11]. Due to the scope of this thesis, I here summarize this history from 1986 onward. In this year, the first self-driving car was introduced by the Navlab team at Carnegie Mellon University (CMU) [1]. The project was funded by DARPA Strategic Computing Initiative. In 1989, CMU pioneered neural networks for autonomous vehicles, which is now considered state-of-the-art. The team demonstrated its prototype with a 5,000 km journey crossing the country in 1995, in which 98% was autonomous.

In Europe, project PROMETHEUS was initiated in 1986, which incorporated more than 13 carmakers and numerous research units from European governments and universities. Nine years later, the car prototype, 95% self-controlled, navigated from Munich, Germany, to Odense, Denmark. However, the car only travelled on highways. Instead, Franke et al. [12] presented the algorithms to control an autonomous car in urban streets, where the authors used stereo-vision for detecting and tracking the vehicles in front of the car and other objects, such as pedestrians, traffic signs and traffic lights were detected and recognized by a general monocular-based framework including detection and classification stages.

In addition to autonomous design and test efforts, commercial cars have gradually been equipped with a driver assistance system. In 1992, Mitsubishi was the first to implement distance-warning using Lidars on the Japanese market. In 1995, they upgraded the warning-system to distance control, whereby the algorithm controls the gear and the throttle to keep an appropriate distance. In 1999, Mercedes introduced the first radar-based ACC. In the same year, Subaru announced the world's first camera-based ACC.

To foster the autonomous cars innovation, the DARPA held the first Grand Challenge in 2004. The participant cars were required to complete a 240 km course in the Mojave desert; in the event, no team was able to finish. In 2005, the challenge was again held again in a desert environment. There were five vehicles that could complete the race. The Stanford team arrived in first position, while the CMU team finished second. The latest DARPA challenge was organized in 2007. This time, the course was 96 km long and in an urban area with traffic lights, signs, and obstacles. The CMU team won the first prize, and the Stanford team obtained the second position. A notable point is that most vehicles that completed the route used Velodyne's multi-beam LiDAR technology. This multi-beam LiDAR is essential for

[1] https://www.youtube.com/watch?v=ntIczNQKfjQ

self-driving cars in urban areas because it can scan 360-degree and provide depth information with obstacles around the car.

Google began their autonomous car program in 2009. They targeted the building of a new driving platform and a multi-beam LiDAR at a reasonable cost. During that time, many big carmakers were testing their self-driving cars; BMW had been testing theirs since 2005. Audi showcased their car's ability to go to the top of Pikes Peak - which is 4,300 meter high - at close to race speeds. General Motors created an autonomous electric vehicle for urban areas in 2011. In 2012, Volkswagen started to test their ADAS system which targeted the ability for a car to drive 130 km/h on the highway.

In 2010, Vislab at University of Parma, Italy, held the 15,900 km test run from Parma to Shanghai Expo in China. In the same year, the Technical University of Braunschweig demonstrated its car's ability to self-drive in a restricted city area. The car was equipped with LiDAR, cameras, and HD maps. It was the first car that had a license for autonomous driving on the streets and highways in Germany.

In 2012, the KITTI benchmark was introduced. It helped researchers evaluate their vision-based algorithms' performance and compare with others'. The specific tasks that can be benchmarked using KITTI is 3D reconstruction, motion estimation, and object recognition. KITTI also released their new benchmark version in 2015.

Since 2012, neural networks for vision processing have been extensively developed and have achieved very high accuracy. Up to now, most state-of-the-art vision-based algorithms on autonomous cars are based on neural networks.

In 2013, the collaboration between Daimler and the Karlsruhe Institute of Technology created the S5000 Intelligent Drive system on Mercedes Benz. The car can fully autonomously drive for about 100 kilometers on the historic Bertha Benz route from Mannheim to Pforzheim, Germany. The car used radar and stereo cameras for object and free-space detection. Traffic lights detection and object classification were processed by monocular vision. Moreover, point-feature-based and lane-marking-based were used for high accuracy localization. In the same year, Nissan demonstrated their autonomous Leaf car for the first time in on the Sagami Expressway in Kanagawa, Japan.

In 2014, SAE International, a society of automotive engineers, published a classification system with six levels for autonomous systems. The levels range from 0 to 5, corresponding to autonomy from fully manual to fully automated. Within this year, Mercedes S Class and Tesla Autoplot introduced their Level 2 models, in which the ADAS system controlled the steering wheel, the throttle, and the brake on the highways. The driver at this level is required to monitor the system at all times.

In 2015, the ridesharing company, Uber, started its program to develop self-driving cars. Also in this year, Delphi Automotive's autonomous car became the first vehicle to finish a coast-to-coast trip from San Francisco to New York. The ADAS system controlled the car 99% of the route.

After six years of developing and testing, Google announced that their self-driving vehicles had been involved in 14 minor accidents. The company was nevertheless optimistic about the project, since the cars have passed nearly 2 million miles on road and all the accidents were caused by humans driving other cars. The project team then became Waymo. Nowadays, Waymo operates a commercial self-driving taxi service in Phoenix, Arizona. It is to date the only self-driving commercial service that self-drive without safety backup drivers in the vehicle.

In 2016, the Tesla model S electric car was involved in a fatal accident while driving in Autopilot mode. It is reported that the car failed to apply the brakes. It is not clear whether the person in car was aware of the situation. Later, in October 2016, the company said that all its vehicles are built with necessary sensors to enable fully autonomous capability, including eight cameras, twelve ultrasonic sensors, and a forward-facing radar. The system will work in the background to collect street scene information for the company to improve its ADAS software. They planned to release their fully autonomous cars by 2017 after millions of miles of testing. However, at this point, their fully autonomous car is not available; in 2020, the company only released a beta version of its fully self-driving software to a small group of testers in the US. In the same year, NVIDIA leveraged a single convolutional network to showcase the ability to self-drive 99% of the route from Holmdel to Atlantic Highlands in Monmouth County, New Jersey.

In 2017, Audi announced their plan to produce their Level 3 autonomous car using 3D LiDAR in addition to cameras and ultrasonic sensors. But in 2020, the company said that this system will not be activated.

In 2018, there was another fatal accident related to autonomous cars. The car is from Uber and it seems that the pedestrian detection system was not working properly.

In 2019, Bosch and Daimler opened shuttle service with their autonomous cars on limited routes in California. In the same year, the European Union defined the regulation 2019/2144 for automated vehicles and fully automated vehicles, applying from 2022. One year later, Japan approved the Automate Lane Keeping System regulation which will be applied in 2021.

In March 2021, Honda started leasing Legend Hybrid Ex cars, installed with the Sensing Elite safety system. The system has the Traffic Jam Pilot function, which was granted the safety certification by the Japanese government. That means drivers in these cars can legally

take their eyes off the roads. The car determines its location and road conditions using 3D high-definition maps and a global navigation satellite system. External sensors include 2 front cameras, 5 radar sensors, and 5 LiDAR sensors.

2.3 Vision-based applications on autonomous cars

2.3.1 Image sensors

Vision is the key input for car navigation. Visual input comes from cameras, which are cheap and easy to install in cars. Key vision-based algorithms consist of semantic segmentation, lane detection, traffic signs detection, traffic lights detection, pedestrian detection, and obstacle detection. Image sensors provide higher spatial resolution for these applications. Moreover, traffic signs and traffic lights recognition require color information, which is only supported by image sensors. Therefore, cameras are the preferable choice for autonomous cars.

There are different technologies for building image sensors. Regarding the semiconductor process, the first image sensors used Charged Coupled Devices (CCD). This technology provides very high image quality, but its manufacturing cost is also high. Nowadays, CMOS is the most commonly used technology to create pixels, arrays (or frames), and imagers. CMOS sensors are cheaper and consumes less power than CCD sensors and also profit from the resolution evolution (Moore's Law) attached to the general CMOS fabrication processes.

Image sensors can be classified based on their acquisition process. For frame-based sensors, all pixels of a frame are captured in a given period of time. The electric charge produced by the light is integrated on a capacitor during the exposure time. There are two exposure modes to sense the light. For the global shutter mode, all pixels are exposed to light at once and integrated simultaneously at that point in time. The second mode is rolling shutter, where pixels are integrated sequentially, row by row and, thus, different rows of pixels are exposed at different time slots, sequentially. Global shutter mode imposes higher latency while rolling shutter is more sensitive to fast motion scenes. On the other hand, Dynamic Vision sensors or event-based cameras do not use the shutter for their acquisition process. Pixels are captured independently and asynchronously. Each pixel keeps monitoring its current intensity and compares it with its intensity in the previous event. If the difference is above a specific threshold, an event is sent out, including the new intensity value and the occurrence time information. When compared to frame-based sensors, they have higher temporal resolution, higher dynamic range, and no motion blur. An event-based camera is

preferable for scenes with minor changes over time, because high event rates consume high power. The vast majority of computer vision research is based on frame-based sensors.

Classification of image sensors can be conducted according to the wavelength of the light they acquire. Consumer sensors typically work in the visible light range (e.g 400-700nm), capturing red, green, and blue bands - the frequencies detected by human eye. Each pixel acquires a single color component by the corresponding color filter placed on top of it. Different pixels get different colors, and they are usually arranged in a Bayer Pattern (2x2 array of adjacent pixels are: 1 green + 1 red and 1 blue + 1 green).

Color information of an image can be represented by tuples of numbers using various color models or spaces. In computer vision, the most used color models are RGB and HSV/HSI. The RGB model produces a specific color by mixing three primary colors red, green, and blue. This model is based on the way human eyes respond to light. However, it is difficult for a human to understand a color from a tuple of RGB values.

HSV and HSI are the models that remap the RGB color model into other dimensions that make more sense to human perception. The names of the models indicate their components. They share two dimensions, H (hue) and S (saturation). Hue is the only dimension indicating color. Hue value ranges from 0° to 360° - with 0°, 120°, and 360° being red, green, and blue respectively. The saturation value indicates the purity of the hue color, in which 100% is the purest and 0% is grayscale. Finally, both V (value) and I (intensity) indicate the level of brightness or luminosity of the hue color.

Some sensors sense intensity only (i.e., monochrome or grayscale sensors being visible, infrared, or X-ray). Despite its limitation in identifying colors, monochrome is installed in real autonomous cars because it helps the car's processor mitigate workload. The opposite approach is used by the sensors that can capture wider wavelength bands, such as Infra-Red or hyperspectral imagers. InfraRed sensors (near, short and long wave length infrared), also known as FLIR, are used by some researchers to detect pedestrians at night time [13]. Hyperspectral data is the fusion of RGB image data and near field and/or far-field infrared data. However, Infrared or Multi-spectral imagers are rarely used in autonomous cars.

Another important factor is the dynamic range. Autonomous vehicles face extreme light conditions, ranging from a very sunny day to night conditions. Most automotive image sensors are designed to work with a high dynamic range (HDR) or wide dynamic range (WDR) that include specific implementations both in hardware (imager) and software (image signal processor), to increase dynamic range by using different technologies such as logarithmic sensors, using higher number of bits per pixel (which required better signal to noise ratio), or combining consecutive images obtained by multiple exposures.

In addition to these common types of camera sensors, there are less common models such as 360-degree or 3D cameras such as stereo, structured light, or time of flight (ToF). A stereo camera system tries to mimic the human eye to obtain depth information from a scene. Similarly, a structured light system is also based on the stereo vision and the triangulation to calculate the depth information. The difference is that the structured light system replaces one camera by a projector that projects known and special patterns onto the observing scene. The frames captured by the camera sensor and the projecting patterns are used to determine the depth. The idea behind structured light is to introduce levels of texture into the scene, to improve the accuracy of a stereo vision system. The ToF camera works in a different mechanism, which generates a modulated light source, usually in the near-infrared range, onto the scene to obtain the reflected light through an image sensor. Images are later analyzed by specific algorithms to understand reality.

2.3.2 Evaluation metrics

Computer vision algorithms extract high-level information from images. That information can range from sequences of images, to a single frame, to regions of an image, or even individual pixels of the image. These different scopes require different computational methods to evaluate their accuracy, so that an objective comparison among them is possible.

Image classification algorithms deal with binary-class or multi-class classification problems. Binary classification problems are typically measured with precision TPR (Eq. 2.1) and recall FPR (Eq. 2.2). Detection algorithms usually result in a confidence value. A detection is considered positive if the confidence value is bigger than a given threshold. In the scope of this thesis, this threshold is referred as classification threshold. However, a positive is only a true positive (TP) if the intersection over union (IoU) between the prediction box and the ground truth is higher than a chosen threshold (e.g. 50%). Otherwise, the detection is a false positive (FP). In this thesis, this threshold is referred to as the IoU threshold. Fig. 2.4 illustrates the IoU between the bounding box (e.g. the prediction) and the ground truth. Ground truth annotations can be provided as a collection of pixel coordinates or a simpler description of an area such as a bounding box.

$$TPR = \frac{TP}{TP+FP} \tag{2.1}$$

$$FPR = \frac{TP}{TP+FN} \tag{2.2}$$

The performance of a binary classifier is often represented by the precision-recall curve. The curve is drawn by varying the classification threshold and achieving corresponding

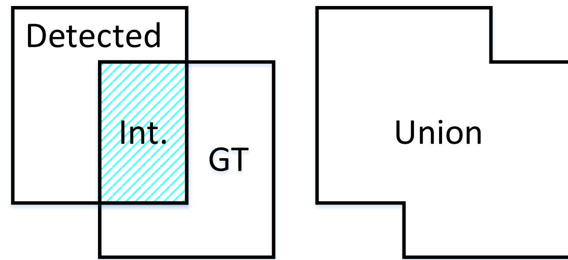


Fig. 2.4 Example of the Intersection of a detected area with a ground-truth recording (left) and the area of their union (right).

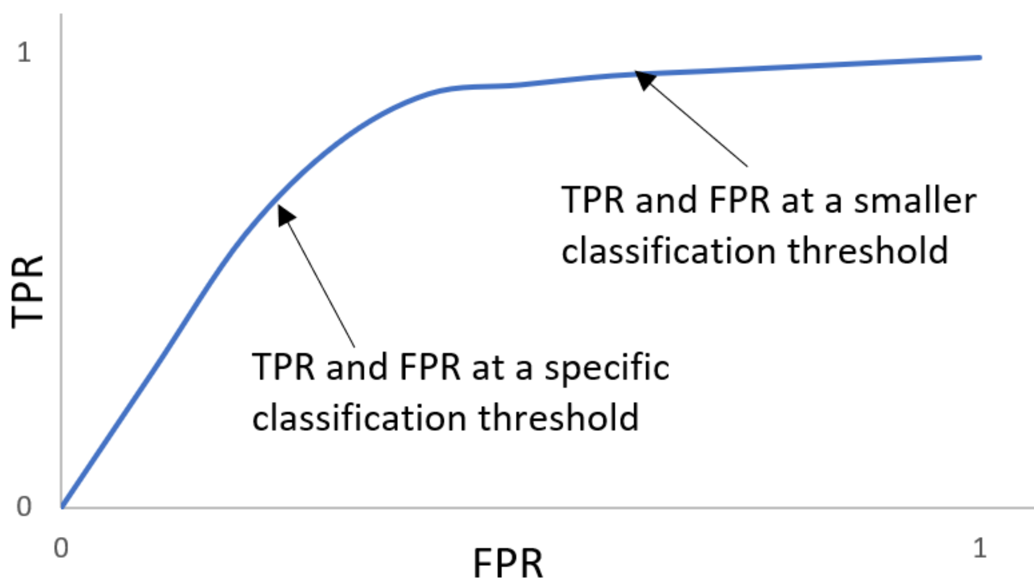


Fig. 2.5 Precision Recall curve example

precision (TPR) and recall (FPR) values. Figure 2.5 illustrates one example of this precision recall curve. A smaller classification threshold would result in more positives and therefore higher FPR and higher TPR. Classifiers can be compared by their precision at a specific recall. However, this precision is only for a specific recall and it does not represent all possible recalls. Average precision (AP) is a better evaluation metric for binary classifier. It is calculated as the Area Under the precision recall curve. The larger the AP, the better classifier.

For multi-class classification, the detection performance is commonly measured by mean Average Precision (mAP). It is the mean of the average precision (AP) of different classes. In the recent well-known dataset, MS-COCO [14], AP is defined differently from the one in binary classifier. It is the average AP over multiple IoUs, from 0.5 to 0.95 with a step size of 0.05.

Another common accuracy metric is F-Measure. This metric combines both the precision and the recall value of a binary of multi-class classifier. Because the precision alone does not reflect the recall value and vice versa. F-measure is calculated as in Eq. 2.3. The parameter β can be chosen to weigh the importance of the recall value. F1 measure corresponds to the case $\beta = 1$, which means the recall and the precision value are equally important.

$$F_{\beta} = (1+\beta^2)\frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall} \tag{2.3}$$

In depth sensing, accuracy is measured at pixel level. In KITTI dataset [15], a pixel is counted as wrong prediction if its disparity is different from the ground truth more than 3. The percent of these outlier pixels over the left image is the key performance metric in the KITTI dataset.

2.3.3 Applications

The vision-based applications in the scope of this thesis are the ones that take input from image sensors. These algorithms are the shaded boxes in Fig. 2.2. The figure is redrawn as in Fig. 2.6 to include only the vision-based functions paths, except for fatigue sensing module (since its input scene is from inside the car and is independent from the street scene, which is out of the scope of this thesis). They can be classified into three different kinds of algorithms, based on the approaches they use to solve the problems. These are depth sensing, semantic segmentation, and object detection. These algorithms are described in the following sections.

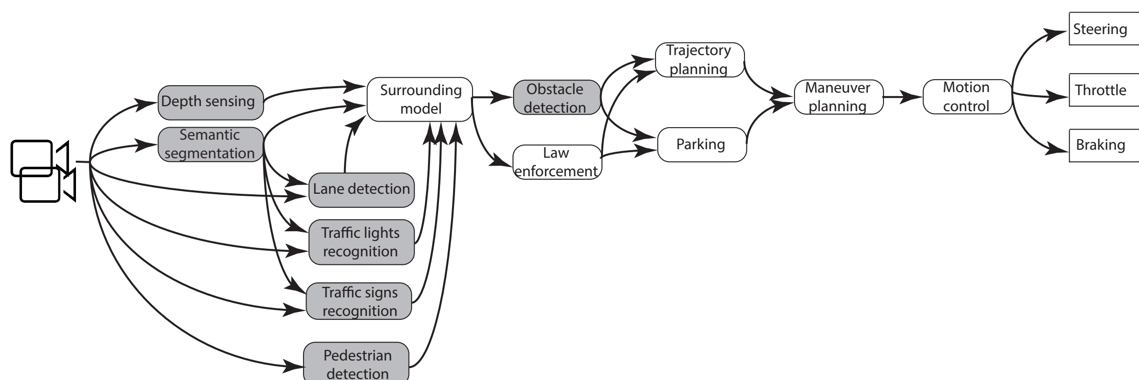


Fig. 2.6 Typical vision-based algorithms of a self-driving car

2.3.4 Depth sensing

The general purpose of stereo matching methods is to infer 3D information (like 3D point reconstruction or depth maps) from two images of a stereo camera that collects images from two viewpoints of the same scenario. In particular, the depth $Z(i,j)$ of each pixel, (i,j), can be calculated as the inverse of the disparity (distance) between image pixels, corresponding to the projection of the same 3D point:

$$Z(i,j) = \frac{Bf}{d(i,j)} \tag{2.4}$$

where B is the distance between the two cameras, f is the focal length of the cameras and $d(i,j)$ is the disparity value of the pixel. The computation of disparity maps [16] can be split in 4 main stages: 1) computation of the matching cost,2) cost aggregation, 3) disparity computation/optimization, and 4) disparity refinement.

Matching pixels are usually obtained by searching horizontally in a disparity range and comparing image features of rectified (i.e., aligned) images [17]. Image features can be either the same image intensity or visual features like edges, Hough or SIFT [18]. Cost aggregation and disparity computation/optimization are closely linked since they involve the computation of the disparity map from the matching cost. Finally, disparity refinement aims to regularize the disparity map, while handling object occlusion where matching cannot be defined. The largest differences across methods concern the implementation of the later steps. In this context, existing approaches can be categorized into local, global and semi-local methods.

Local methods aggregate the matching costs in support regions surrounding each pixel (usually using a weighted average) [18]. The disparity value that gives the lowest difference between support regions is set as the disparity value of the center pixel of the support region in a winner-takes-all approach (WTA).

Global approaches formulate aggregation and disparity computation as a global optimization problem [18]. The disparity map is computed as the minimum of a functional defined as a weighted sum of the matching cost of all pixels and a regularizing penalty term that encourages neighboring pixels to have similar disparities.

Local algorithms prioritize speed over accuracy, while global algorithms are time consuming but very accurate. Semi-local approaches [19] have recently become the best compromise between accuracy and speed. Semi-global approaches efficiently compute a global map from the initial local costs by simplifying the smoothness costs so that it can be optimized along different directions separately.

Generation of depth-maps could be considered low level CV applications that are required for more complex subsequent functions like obstacle detection. CV methods compete with

LIDAR as a more economical method to build depth-maps. In vision-based depth sensing, depth-maps can be created by combining the information from multiple cameras. In the most typical case, two cameras are combined, and depth-maps are created by stereo-matching, i.e., for each region in one image from a camera, its corresponding area is found in the same epipolar plane of the image acquired by the other camera. The distance between these related areas is known as disparity and can be used to infer the distance of the area. An example is illustrated in Fig. 2.7, where the left image is on top, the right image is in the middle, and the depth-map is at the bottom.

Depth maps can also be inferred from monocular cameras. However, the accuracy of monocular depth sensing is still not comparable to that achieved by stereo vision [20].

KITTI benchmark [21] is the most popular benchmark for depth sensing algorithms on autonomous cars. Benchmarks are typically annotated with labels coming from LiDAR, which has better accuracy. The metric used in KITTI is D1-all which means the percentage of disparity outliers in the first image (e.g. left image) over all the ground truth pixels. A disparity is considered an outlier when its error is equal, or more than 3 pixels compared to the ground truth. With respect to this metric, the state-of-the-art method is presented in [22], with code. The authors use a neural architecture search to obtain a feature net and matching net for the stereo matching pipeline. It achieves 1.65% of D1-all and a throughput of 3 FPS. Table 2.1 presents the common datasets for stereo matching algorithms and their SoA implementations.

Table 2.1 Stereo depth estimation datasets

Dataset	Year	Best Result	Metric	Score
KITTI 2012 [24]	2012	LEAStereo [22]	Out-Noc	1.13%
Middlebury 2014	2014	LocalExp [25]	Avg	5.43
KITTI 2015	2015	LEAStereo [22]	D1-all	1.65 %

2.3.5 Semantic segmentation

Semantic segmentation is the algorithm that assigns every pixel in the input image a label among a known set of class labels. The classes can be, for example, pedestrian, cyclist, road, sidewalk, traffic sign, traffic light. Multiple objects of the same class are considered a single entity and annotated by the same color as shown in Fig. 2.8. The algorithm that separates not only the classes but also the objects of the same class is instance segmentation. An example of instance segmentation is illustrated in Fig. 2.9. It is similar to object detection because it requires both classification and localization. The difference is that object detection locates

Fig. 2.7 Example of the disparity map computed using SGM from stereo images provided in [23]

and recognizes instances in bounding boxes while semantic segmentation separates classes of object at pixel-level granularity.

Fig. 2.8 Semantic segmentation example Cityscapes dataset [26]. Objects in the same class have the same annotation color.

Surveys on the algorithm targeting autonomous driving application are published in [28], [29]. There are different strategies to build a semantic segmentation system. The input is the images from cameras. A high definition (HD) map, such as semantic point cloud map, can be fused to improve the system's accuracy. Moreover, Lidar sensors can be integrated to provide depth information which helps to localize objects and segment the scene. The two common dataset for semantic segmentation are Citiscapes [30] and Camvid [31]. Mean IoU is commonly used to indicate the accuracy of a semantic segmentation system. It is the mean IOU of all the classes of the dataset. The higher mIoU, the more accurate. The SoA methods on available benchmarks are listed in Table 2.2.

The best implementations in accuracy are based on CNNs. The CNN models for semantic segmentation are variants of Unet [32], Fully Convolutional Network (FCN) [33], Segnet [34], and Pyramid Scene Parsing Network (PSPNet) [35]. The SoA model uses DeepLabv3 network [36], which combines the advantages of Unet and Segnet. It achieves the highest mIoUs of 82.9% on Camvid and 72.8 on KITTI.

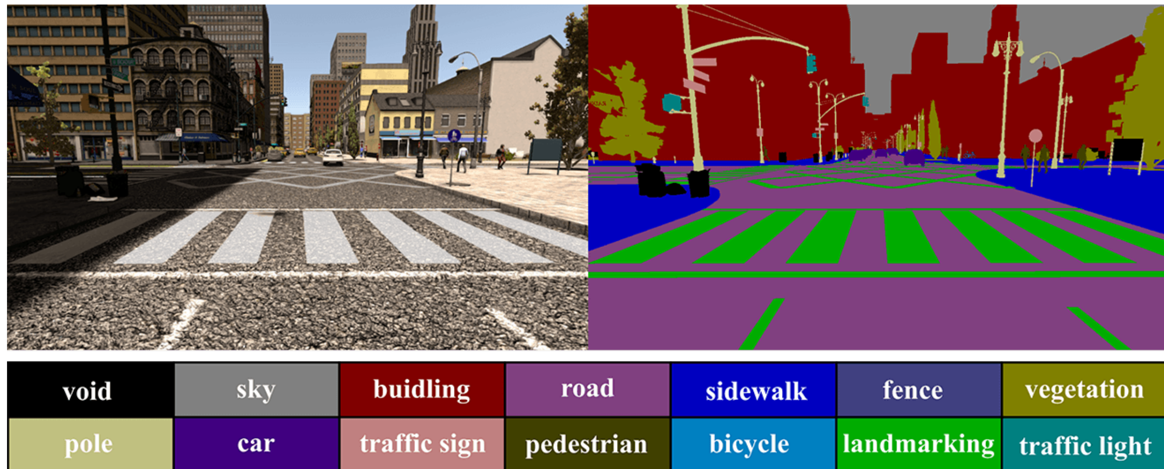


Fig. 2.9 Instance segmentation example from the Synthia synthetic dataset [27]. Each person has a unique annotation color.

Table 2.2 Semantic segmentation datasets

Dataset	Year	Best Result	Metric	Score
Pascal VOC2012	2012	EfficientNet-L2+NAS-FPN [37]	mIoU	90.5 %
Camvid [38]	2009	DeepLabV3Plus + SDCNetAug [36]	mIoU	82.9 %
Cityscapes [26]	2016	HRNet-OCR [39]	mIoU	85.1 %
Kitti [15]	2018	DeepLabV3Plus + SDCNetAug [36]	mIoU	72.8

2.3.6 Object detection

The detection algorithm recognizes objects of certain predefined classes (pedestrians, vehicles, road objects such as lane, traffic lights, traffic signs, buildings, animals, etc.) in an image and locates their positions (e.g. using bounding boxes). Due to safety-critical reasons, lane detection, TSR, TLR and pedestrian detection are often implemented as stand-alone applications. Obstacles can be other vehicles, cyclists, road structures - such as roud boundaries, trees, walls, buildings. Fig 2.10 presents an example of object detection on a street scene. It comes from the detection result published in [40]. The COCO dataset provides ground truth at pixel-level granularity, which benefits instance segmentation algorithms. However, the object detector usually uses bounding boxes to locate the prediction.

Popular dataset are: KITTI [21], PASCAL VOC [41], and MS-COCO [14]. The state-of-the-art methods for implementing vision-based object detection are mostly based on DNNs. In [42], Girshick et al. are the firsts to show that CNN-based object detector provides higher detection performance than HOG-based detector on PASCAL VOC dataset. The authors propose regions that potentially contain objects and then classify all the regions by

Fig. 2.10 Detection result example on COCO dataset done in [40]. The ground truth objects at pixel granularity and the predictions are bounding boxes

category-specific classifiers. This method is categorized as two-stage object detection [43], [44]. On the MS-COCO dataset, the state of the art two-stage algorithm is Mask R-CNN [40].

The typical steps of a two-stage method are generalized in Fig. 2.11. The pre-processing step can be image scaling since image resolution used by the algorithms depends both on available computation and real time-constrains. The first stage proposes the regions that might contain objects. The second stage is a CNN that extracts features from the regions of interest (RoI). The features are used for classification and bounding box regression. The Mask R-CNN also uses the features for instance segmentation. This method provides the highest accuracy on the MS-COCO [14], a well-recognized dataset.

Regarding real-time efficiency, a one-stage approach is more suitable. It does not have the region proposal step [45]. The state-of-the-art method is YOLOv3 [3] with a throughput of 19 FPS (input image is 608x608). It uses Darknet-53 to extract features from the full image. YOLOv3 is 3.8x faster than the RetinaNet model but its accuracy is slightly lower.

The RetainaNet model incorporates ResNeXt and Feature Pyramid Network [46] to achieve the state-of-the-art one-stage method in terms of accuracy on MS-COCO dataset.

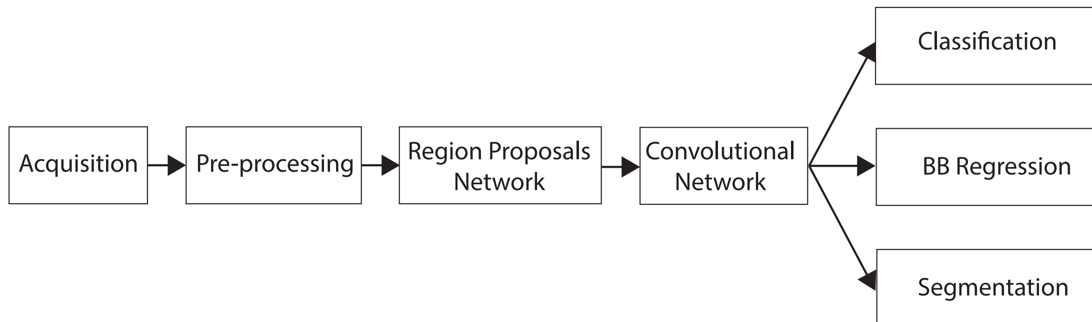


Fig. 2.11 Typical processing steps of a CNN-based object detection algorithm.

In the next sections, sub-problems of object detection on autonomous cars are described. They are lane detection, TSR, TLR, and obstacles detection. Pedestrian detection, the research goal of this thesis, will be the focus of the next chapter.

2.3.7 Lane detection

Lane detection (LD) is an algorithm that detects and extracts lanes on roads at a distance of several tens of meters ahead [47]. It is the key algorithm in many other applications such as lane departure warning systems (LDWS), advanced cruise control (ACC), lane keeping assist (LKA), lane centering assist (LCA), lane change assist, and turning assist. Vision sensors are likely used in lane detection systems, since lane marks are created for human vision. Lane detection systems must be able to detect lane marks on the road surface, infer the drivable lanes, and determine the current lane (ego lane).

Datasets for LD are available, with some focusing on the recognizing of lane marks and others focusing on the recognition of the whole lane area. For most, the ground truth is provided at pixel level, but in some cases the ground truth is provided as a mathematical function that describes the lane lines. Fig. 2.12 gives an example of a challenging lane detection scenario.

The problem can be presented as a binary class segmentation if only lane marks are considered, or multi-class segmentation if other road marks are considered; or even instance segmentation if different lane marks are considered. Table 2.3 lists some available datasets for lane detection, together with current best performing designs.

State-of-the-art methods to implement lane detection are based on CNNs. In [51], Spatial CNN (SCNN) is proposed to tackle the long, continuous structure of lane markings. The

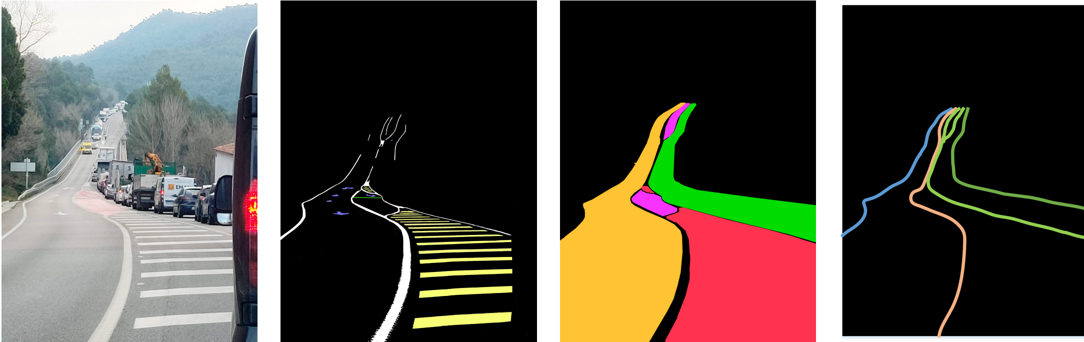


Fig. 2.12 Fragment of a challenging scenario for lane detection with a corresponding pixel level ground-truth for visible lane marks (center), and partially ocluded road lanes (right).

Table 2.3 Lane detection datasets

Dataset	Year	Best Result	Metric	Score
Caltech [48]	2008	HT+FCN	F_1[a]	96.29 %
TuSimple [49]	2017	FOLOLane [50]	TPR[b]	96.92 %
CULane [51]	2018	CondLaneNet [52]	F_1	79.48 %
LLAMAS [53]	2019	LaneAF	F_1	96.0 %
BDD100K [54]	2019	ERFNet+ESA [55]	TPR	60.24 %

[a] Metric for pixel-based evaluation calculated from precision and recall values, as described in section 2.3.2.
[b] It is the ratio of true positives above true positives plus false negatives, described in section 2.3.2.

rule of thumb is to reinforce spatial information via inter-layer propagation of the CNN. The implementation achieves an accuracy of 96.53% with the TuSimple dataset [49]. The state-of-the-art accuracy on TuSimple dataset is reported in [50], with an accuracy of 96.92%. The authors propose FOLOLane, which uses CNN to predict the key points in the local range and correlates ones of the same lane line.

Regarding Caltech dataset, the state-of-the-art accuracy is 96.29%, published in [48]. The authors combine HT and a deep convolutional network based on the Fully Connected Network on GTX 1070.

Another approach for lane detection is presented in [56]. The authors perform instance segmentation using LaneNet, and the resulting pixels from each instance are used for polynomial curve fitting.

2.3.8 Traffic signs recognition

Traffic sign recognition (TSR) is the process of detecting traffic signs in an image and recognizing their meaning. Like LD, TSR is one of the applications that tend to employ image sensors, because colors are very important to sign meaning; for instance, red is typically associated with prohibitory signs. There are many challenges to achieve a high recognition rate, such as the effect of lighting conditions, motion blur, fading of signs due to environmental conditions, rotation, and occlusion [57]. Fig. 2.13 illustrates some cases in which TSR needs to detect and recognize traffic signs.

Surveys on vision-based TSR are presented in [58], [59]. The authors in [59] point out that different countries still have different designs for the same traffic signs, which imposes greater computational load into the TSR system. Some datasets for TSR systems are The Belgian Traffic Sign Dataset [60], The German Traffic Sign Recognition [61] and Detection Benchmark [62], Swedish Traffic Sign Dataset (STSD).

The detection performance is commonly measured by mean Average Precision (mAP), as described in Section 2.3.2. The state-of-the-art implementations on GTSDB and STSD datasets are shown in Table 2.4. The TSR system in [63] is only for inventory management applications. It achieves the SoA miss rate of 3.5% on Swedish Traffic Sign Dataset (STSD) using Mask R-CNN [40].

On GTSDB dataset, the SoA model [64] achieves 97.71% mAP. The detection module uses Mask R-CNN network and a CNN architecture is proposed to classify the signs.

Table 2.4 Best traffic sign recognition implementations on GTSDB and STSD

Dataset	Best Result	Year	Metric	Score
GTSDB	MASK_R-CNN + Class_CNN [64]	2020	mAP	99.71 %
STSD	Mask R-CNN [63]	2019	mAP	96.5 %

2.3.9 Traffic lights recognition

Vision-based TLR detects traffic lights in the input images and recognizes the state of the light signal. Fig. 2.14 illustrates a case of traffic lights detection. Surveys on this application can be found in [65], [66]. One popular approach is to extract features from the input images, like colors and shapes, and classify the objects based on the features. Recently, most of the best algorithms use DNNs. Table 2.5 presents the most accurate implementations on some popular datasets for TLR. The authors are evaluating TLR with different accuracy metrics.

Fig. 2.13 Multiple signs on both sides at the far sight including information, warning, and prohibitory signs. b) Two attaching prohibitory signs and an information sign with a secondary sign attached.

2.3.10 Obstacle detection

Obstacle detection is the application that segments on-road generic obstacles from free-space in the front area of the vehicle. Obstacle detection is the foundation from which to implement more complex systems, such as collision avoidance, situation analysis, cruise-control, and path planning. A camera-based system could bring higher spatial resolution than other active sensor systems such as laser, radar, and ultrasonic. Fig. 2.15 shows an example of obstacles in a street scene. There are two vehicles on the left lane and one inexplicable object on the ego-lane.

The most common approach when using vision sensors is to construct a 3D model of the scene. A survey on real-time obstacle detection has categorized these systems into four different models based on the clustering strategy [75]. All the models employ stereo images

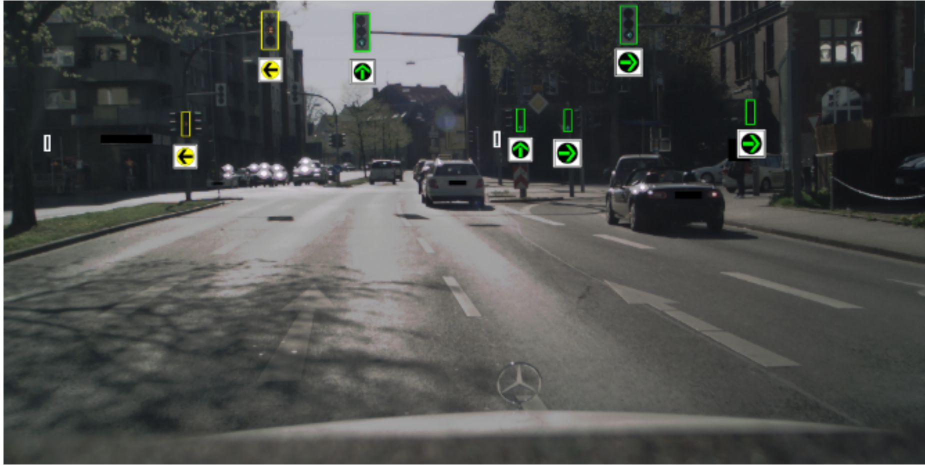


Fig. 2.14 Traffic lights detection example in Germany [67].

Table 2.5 Traffic light recognition datasets and their best implementations

Dataset	Year	Best Result	Metric	Score
LaRA [68]	2010	Feature learning - Fusion detection [69]	AUC	88.61%[a]
LaRA [68]	2010	2D CNN [70]	mean F1	99.96%
LISA [71]	2015	2D CNN [70]	mean F1	99.92%
Bosch [72]	2017	2D CNN [70]	mean F1	89.44%
DriveU [73]	2018	Single Shot Detection [74]	LAMR	1.5%[b]
DriveU [73]	2018	Fast-RCNN + ResNet [67]	AP_{50}	92%[c]

[a] The highest average between AUCs of red and green lights is chosen.
[b] Log average miss rate.
[c] Ignoring all the less than or equal 8 pixels bounding boxes.

and disparity maps. The accuracy of a stereo-vision system highly depends on the camera calibration. In [76], four different configurations regarding the focal length and the distance between cameras are explored to achieve a detection range from 10 to 140m.

Based on the assumption that the disparity should be equal or smaller while going from bottom to top in any specific column of an image, an obstacle can be detected only by analyzing the disparity map. Nevertheless, this simple approach could not handle noisy data from disparity maps. In [77], the authors proposed the detection of free space on road scenes based on occupancy maps [78]. In an occupancy map, each cell represents the likelihood of occupancy at its coordinate and, ideally, only objects lying above the road are registered as obstacles. For global optimum, the dynamic programming technique has been used to obtain the line separating free space and obstacles on a road. The heights of obstacles are also determined by the same dynamic programming.

A popular technique to detect moving objects is optical flow, also known as scene flow [75]. The algorithm tracks movement by comparing temporally continuous images. The optical flow algorithm has a higher computational cost than stereo matching, since it searches for corresponding points both horizontally and vertically. It is usually fused with stereo matching to improve the robustness of the obstacle detection algorithm. This combined technique is referred as 6D-vision [79], in which velocities of moving pixels are tracked. Because of the high computational complexity of tracking 3D pixels, [80] proposed to represent an image using stixels instead of pixels. Stixels are rectangular sticks representing the objects on the street, which are usually vertical. Stixel offers a significant reduction of data volume.

Obstacle detection is also a perfect application for deep-learning based models. It can be considered a sub-problem of object detection, since obstacles can be some certain classes of objects. Best object detectors are based on CNNs. YOLOv3 [3] is the SoA approach which has both good accuracy and high processing speed. RetinaNet [81] and Single Shot MultiBox Detector (SSD) [82] are competitive in accuracy (MS-COCO dataset [14]) but they are three to four times slower.

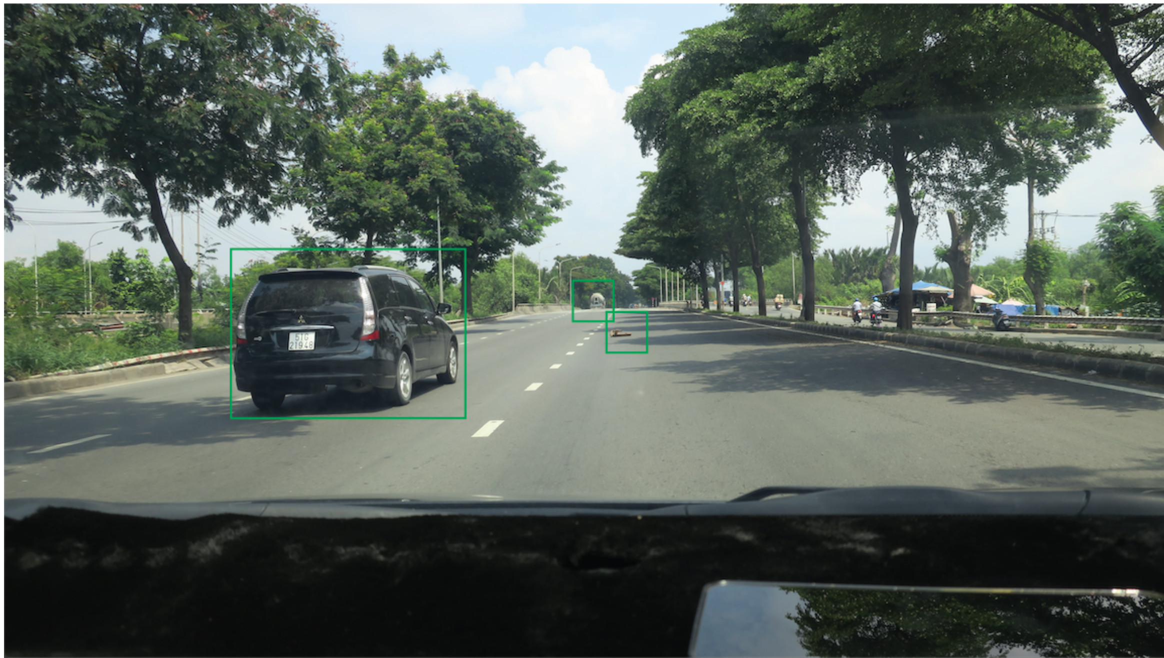


Fig. 2.15 Example of obstacles on a road.

Obstacle detection can be incorporated with semantic segmentation, like in [83]. The authors use Cityscapes [26] and Lost and Found [84] datasets for evaluation. Cityscapes is mainly for semantic segmentation applications. The Lost and Found dataset focuses on

obstacle detection, with fine-grained annotations of small obstacles on roads. The evaluation method of the paper is similar to that of the semantic segmentation problem. The paper achieves 72.5% mIoU on Cityscapes and 72.2 mIoU on the blended dataset.

2.4 Hardware platforms for autonomous cars

Computing hardware platforms for autonomous cars use ASICs, GPUS, CPUs, and FPGAs as accelerators. A range of automakers and their computing hardware platforms are presented in Table 2.6. Some companies have their own ASICs for the computing on the edge. These ASICS are specialized hardware for vision processing algorithms. Waymo seems to use Intel's CPU. Daimler and Uber use a GPU platform from Nvidia. With the increasing demand of workload and real-time processing speed, FPGAs provide an acceleration solution. The technology supports high speed processing with low power consumption in comparison to CPUs and GPUs.

Table 2.6 Automakers and their computing hardware platforms

Announced Year	Company	Computing hardware platform
2017	Waymo	Intel CPU
2018	Daimler and Bosch	Nvidia Drive Pegasus GPU
2018	Mobileye	EyeQ5 ASIC + Intel Atom
2018	Uber	Nvidia Drive PX GPU
2019	Baidu	Baidu BIE-AI-Board, BIE-AI-Box
2019	Tesla	Tesla FSD computer

2.5 FPGA technology

Despite their humble beginnings as a method to implement glue logic to connect more important chips on a PCB, nowadays FPGAs are considered fully-fledged computing platforms and a valid implementing option for high-end cars - despite their currently high device costs. The spatial computing paradigm can profit from FPGA architectures for many highly parallel applications that have an important dataflow component, or high memory locality [85].

Not all applications can benefit from full FPGA implementations but, even in many of these cases, FPGAs can still offer some advantages when used in conjunction with host CPUs as accelerators. The speedup provided by such accelerators is often limited by the sequential part of the algorithms running in the CPU, as reflected by Amdahl's law [86].

Device density is a determinantal factor of the performance and energy efficiency achieved in FPGA design [87]. FPGA manufacturers are early adopters of new technology nodes; therefore, benefiting from their latest advances in higher transistor densities and lower power consumption.

FPGAs are built by replicating many simple basic logic elements (LEs) and some more complex IPs, such as dual-port memories and digital signal processing (DSP) modules. The main components of LEs are Look Up Tables (LUTs) and registers, while the main components of DSPs are adders and multipliers (see Figure 2.16).

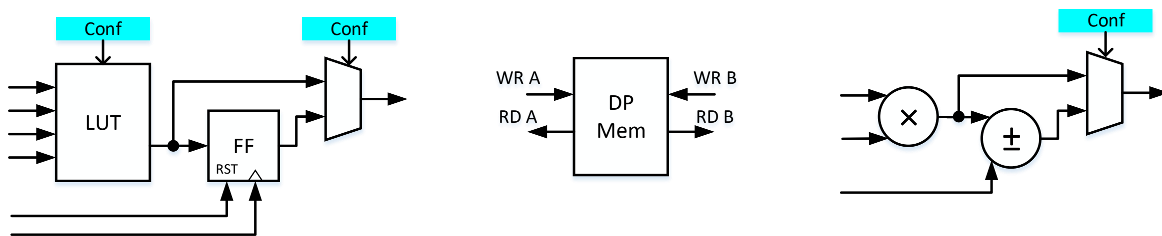


Fig. 2.16 Main replicated FPGA components interconnected by a configurable interconnect.

In the late nineties, FPGAs already had thousands of LEs. To put this in perspective, a simple 32 bits Reduced Instruction Set Computing (RISC) processor needs a few thousand LEs to be implemented. With the new millennium, it became possible to configure a subset of logic blocs to work as a processor and use the rest of the available logic to implement other processing units or system interfaces that could communicate among them and with the processor using buses or other simple mechanisms (directly, dual-port memories, etc.). Nowadays, the logic density is high enough to implement hundreds of simple processors in the same device. Internal memory can exceed the order of several dozens of MB [6]. The availability of low-latency local memories is very important for many computer vision applications. The theoretical peak performance of latest FPGAs can be up to several TFLOP/s. Although these estimations are based on assuming a full usage of all available computing resources, some works have successfully reached the TFLOP/s range [88].

2.5.1 Design methodologies

Applications on autonomous cars usually require high-speed processing. Vision-based applications usually require hardware-software co-design methodology because some tasks are computationally intensive and need to be accelerated using dedicated hardware.

Typical structures of an FPGA-based accelerator are shown in Fig. 2.17. With the structure in Fig. 2.17a, the FPGA and the CPU have separated dedicated memory. The host memory can only be accessed by the host CPU. However, the device memory is accessible to

both the FPGA device and the host CPU. Another structure is illustrated in Fig. 2.17b, in which the host memory is shared between the host CPU and the FPGA device. This structure avoids copying data between the host and the device memory. However, it requires high speed communication bus between the host and the FPGA. For example, Intel packages their Xeon and FPGA die in the same package and the two dies transfer data using two PCIe Gen3 x8 and one QPI physical links. Finally, the structure in Fig. 2.17c describes the case where the host CPU is a soft-core CPU mapped on FPGA fabric. In this structure, the device memory is the only external memory for the whole system.

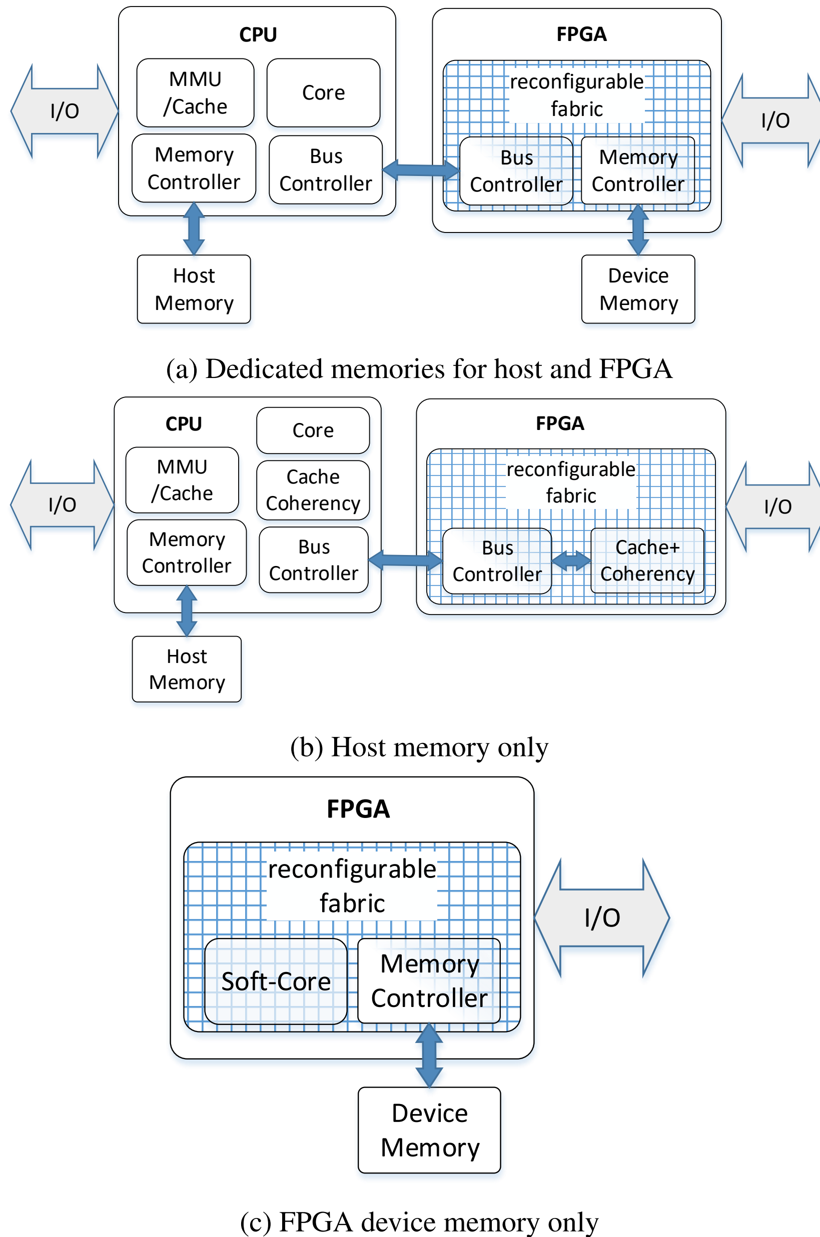
The HDL design flow involves specifying the hardware microarchitecture; describing finite state machines (FSM), pipeline stages, defining registers or bits, etc. This flow requires multiple years of experience on hardware design and, more importantly, huge efforts to verify the hardware function. Writing HDL is sometimes a tedious and error-prone process.

High-level synthesis (HLS) and System-level Synthesis (SLS) offer designers an alternative way to implement their designs onto FPGAs, which helps shortening design and verification time. From a software perspective, this design flow abstracts most of the above mentioned details which require hardware design experience. Hardware designers could also take advantage of using HLS, especially at system level design for design space exploration and verification. Outputs of HLS/SLS tools are the design described in hardware description language (HDL) or System C ready for further steps (back-end or Low-level synthesis). The input specification of an HLS tool depends on the specific compiler. In the case of VivadoHLS, a Xilinx's tool, the input design could be written in C/C++ or SystemC. On the other hand, OpenCL compiler from IntelFPGA accepts OpenCL language as the input. Many other industrial and academic HLS tools could also be found in [89].

2.6 FPGAs for vision-based algorithms

Computer vision techniques commonly relied on feature extraction, either feature classification or regression and, optionally, model verification - as depicted in Fig. 2.18. Images are pre-processed after acquisition. The pre-processing steps can be Bayern pattern filter, noise removing, resizing. The Region of Interest Selection, Feature Extraction, and Classification/Regression blocks can be implemented as modular-based method using classical machine learning techniques. These techniques are suitable for small data and battery-powered problems. It leverages image features such as color, shape, HOG, SIFT, etc.

However, this approach usually gives less accuracy than the SoA approaches using neural networks [90]. Two-stage networks R-CNN [40] include the Region Proposal network which is the same as Region of Interest Selection module. On the other hand, one-stage networks



(a) Dedicated memories for host and FPGA

(b) Host memory only

(c) FPGA device memory only

Fig. 2.17 Typical structures of an FPGA-based accelerator

exclude the Region Proposal module and use a single neural architecture (e.g., CNN) to accomplish both feature extraction and classification/regression [3]. A survey of computer vision algorithms and related hardware implementations is presented in [43] with a special focus on CNNs.

CNN-based implementations typically require both high computing resources and memory bandwidth. This also means they consume high amounts of power. However, energy is a precious resource on autonomous cars. Therefore, the implementations of the algorithms

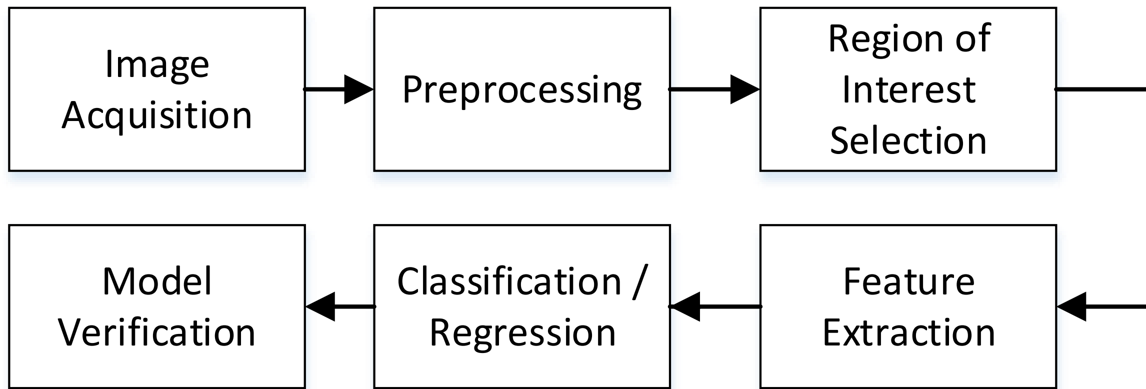


Fig. 2.18 CV generic pipeline

should be energy efficient, in terms of performance per watt. According to [91], an ideal ADAS system should be able to offer a computing speed of over 200 GOPS with no more than 40 W, which is 5 GOPS/W. Matsubara et al. [92] sets an even smaller power budget for the SoC of the system. They proposed that Level 3 cars should achieve 120 TOPS at 12 W, which is 10 TOPS/W. Table 2.7 presents the performance of some hardware platforms for autonomous cars. From the table, we can see that FPGAs achieves several hundreds of GOPS/W power efficiency. GPU platforms give more than 1 TOPS/W, and ASICs can provide up to 10 TOPS/W.

Figure 2.19 reports the performance of these platforms based on several implementations. The power efficiency is represented by the diagonal lines. CPU platform is below 1 GOPS/W. FPGAs are better, in the range of 100 GOPS/W. GPUs achieve around 1 TOPS/W. Finally, ASIC performance spreads in a broad range from 0.1 TOPS/W to 10 TOPS/W.

These are based on the peak power efficiency on neural network computations. According to those numbers, GPUs are 10 times more efficient than FPGAs. However, the experiment results from research papers shown in Table 2.8 indicate another perspective. The papers compare the performance between FPGAs and GPUs using the same algorithm. Based on the reported results, the energy efficiency is extracted and compared between FPGAs and GPUs. In particular, the energy efficiency is measured by the energy to process one frame. It is ratio between power consumption and throughput (FPS).

Table 2.7 Comparison of different hardware platforms for autonomous cars

Platforms	Architecture	Performance (TOPS)	Power (W)	Power eff. (TOPS/W)
NVIDIA Jetson Xavier NX	GPU	21	10	2.1
NVIDIA Jetson AGX Xavier	GPU	32	10	3.2
Renesas SoC [92]	ASIC	60.4 TOPS	4.4	13.8
Google TPU v3	ASIC	4	2	2
Xilinx ZCU102	FPGA	3.69	23 [a]	0.16
Intel® Stratix® 10 NX	FPGA	183	225	0.63

[a] https://developer.xilinx.com/en/articles/accurate-design-power-measurement.html

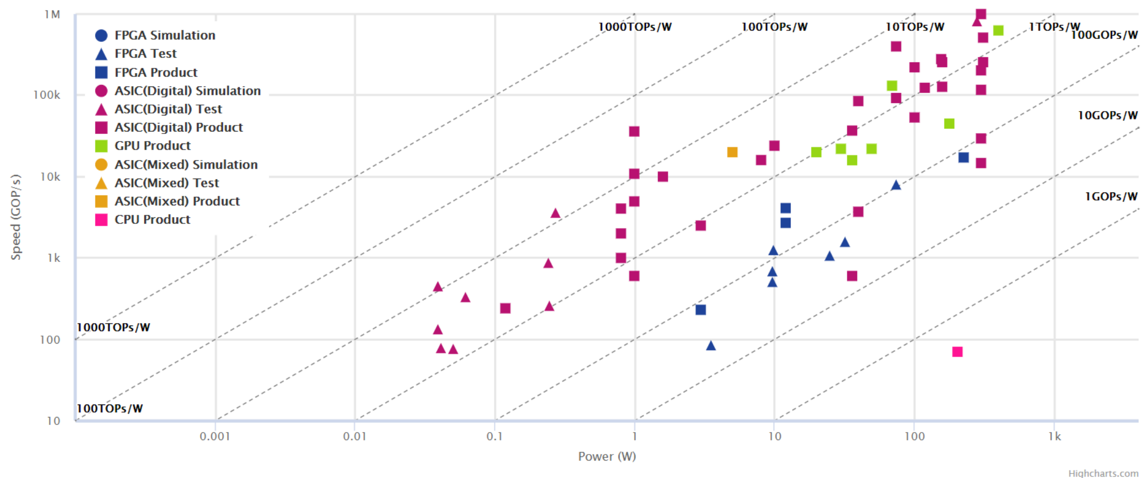


Fig. 2.19 Neural network accelerator platforms [6].

Table 2.8 Energy efficiency of algorithms on FPGAs and GPUs

Ref.	[93]		[94]		[95]		[96]		[97]	
Year	2017		2018		2015		2018		2020	
Algorithm	Sign recognition		Stereo matching		HOG		YOLOV2		BNN Stereo mat.	
Resolution	1920x1080		1242x375		640x480		224x224		1242x375	
Platform	FPGA ZC706	GPU Grid K520	FPGA ZC706	GPU GTX Titan X	FPGA Virtex-6	GPU Tesla K20	FPGA ZCU102	GPU PASCAL	FPGA Stratix V	GPU Xavier
FPS (MHz)	126	41	72	167	68	13.4	40.81	1.48	114.08	8.53
Power(W)	5.5	72	3	250	37	225	4.5	7	3	12.91
Energy eff. (J/frame)	0.04	1.76	0.04	1.5	0.54	17	0.11	4.73	0.03	1.51
(FPGA/GPU)[a]	43		36		31		43		58	

[a] Ratio between energy efficiency (J/frame) of GPU and FPGA

From the Table 2.8, we see that the energy efficiency of FPGAs is from 31x to 58x better than GPUs. It is worth noting that the applications are different and the platforms used are selected independently by the authors. Two out of five papers use CNN-based methods. The other papers use classical machine learning algorithms.

The numbers from Table 2.8 suggest that the energy efficiency is not only dependent on the power efficiency that the hardware platform provides but also on how efficient the hardware platform is being used. The following section describe some common techniques used on FPGAs for vision-based algorithms.

2.6.1 FPGA common techniques

To determine what have been accomplished in the literature, I have conducted a survey on FPGA implementations for vision-based algorithms. The common ways to boost a design's performance is to parallelize its tasks. This can be done by a single or multiple pipelines. Secondly, FPGA on-chip memories are flexible and can be implemented as different types of memory - such as registers, FIFOs, line buffers, and Look-up Tables. Dual port memory on FPGAs helps to simultaneously read and write at the same clock cycle, which partly mitigates memory dependency due to multiple memory access at the memory location at the same time. Finally, fixed-point representation is an advantage of FPGAs in some algorithms when fixed-point numbers provide comparable accuracy to floating-point numbers. Computation of fixed-point numbers is simpler and requires lower latency than that of floating-point calculations.

Custom Memory Buffers

The flexibility of FPGA allows for the implementation of memory buffers addressing the needs of applications. For in line buffers, the goal is to act as a custom cache memory; the memory access pattern is regular, with high data locality. The trick is that, in long shift register we can predict the exact positions where the recently accessed data is located. This therefore avoids many costly external memory accesses since the values are already available in the buffer. The line buffers are depicted in Fig. 2.20 as an illustration.

Task parallelism

The aim of task parallelism is to divide a function into small taks and devote an independent computing system to each.

Since each computing system could be devoted to a single task, this supposes a great opportunity to optimize the performance and the resource for each task.

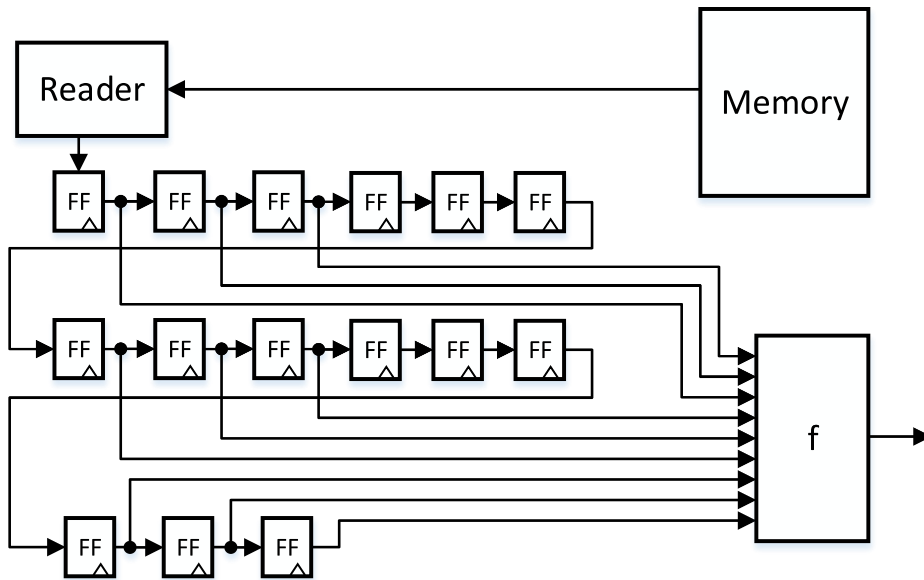


Fig. 2.20 An illustration of line buffer.

Pipelining is a form of task parallelism. It can be applied at several granularities in FPGAs. Pipelining requires storage between the computing elements.

Fig. 2.21 illustrates task parallelism. At fine-grain level, the nature of FPGA LEs (see Fig. 2.16, which already contain registers for storage, makes it highly adequate for implementing pipelines.

At the coarse grain level, dual-port memories also help to store the intermediate values of task level pipelining.

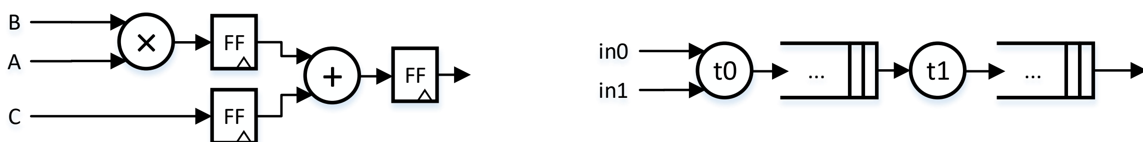


Fig. 2.21 Example of task parallelism at the fine-grain (left) and a coarse grain (right) levels

Data parallelism

When working with independent data, there is the chance of creating multiple computing units that work with several independent sets of the data simultaneously.

If the data storage is centralized, it makes sense if the cost of computing is high enough that the data non concurrent access represents a tiny fraction of the computing time; otherwise,

the speedup of replicating hardware is not worth the cost. A way to minimize this problem is to work with intermediate independent memories (when part of a pipeline).

Number Representation

Floating point numbers are generally used in algorithms because they offer a flexible way to background the valid number range of numbers. The flexibility is balanced against a higher latency. This is an important drawback for iterative algorithms. On the other hand, basic fixed point numbers operations (add, sub, mult) are commonly implemented in single cycle designs. FPGAs support DSP blocks which help to speed up integer multiplication and addition. Fixed-point operations can benefit these DSP blocks, since they are mostly all integer operations. Therefore, moving an algorithm to a fixed-point has two benefits: better performance and less resource usage.

Lookup tables

Many complex mathematical functions can be optimized by having pre-computed values that are stored in on-chip ROM or block RAMs. Therefore, the pipeline just needs to look up instead of computing complex mathematical operations. ROM is sufficient for implementing small lookup tables and it returns result asynchronously and immediately. Larger lookup tables can be implemented in block RAMs, which typically require one clock cycle latency. CORDIC is one algorithm based on Lookup tables.

Chapter 3

Pedestrian detection on autonomous cars

3.1 Pedestrian detection

Pedestrian detection is a safety-critical application on autonomous cars. Only in the US, in 2019, there are more than six thousands pedestrian fatalities, which is 17% of the total traffic deaths [98]. The algorithm can be implemented as a part of object detection. But because of its importance, it is usually considered separately.

SoA approaches for pedestrian detection are based on monocular vision. The algorithm detects pedestrians in images and draws bounding boxes surrounding them. Fig. 3.1 presents some examples of annotated pedestrians, derived from Caltech Pedestrian dataset.

The key challenges of this algorithm are the occlusion, which is usually the case on streets, and the difference in pedestrian's posture. For example, in the top left image of Fig. 3.1, the pedestrians are partly occluded by the trolleys.

Popular datasets for pedestrian detection algorithms are Caltech [99], INRIA [100], Daimler [101], ETH [102], TUD-Brussels [103], CityPersons [104], and EuroCity Persons [105]. They are listed in Table 3.1.

- The INRIA dataset contains a total of 1774 (64x128) images of humans cropped from personal photos. These are used as positive samples for training and testing process. All the annotations are at least 100-pixel height. The negative samples are extracted from 1671 person-free photos. The photos of this dataset are usually high resolution (e.g. 2592x1944). However, the photos are not taken from street scenes.

- In ETH dataset, the photos are obtained from a pair of cameras mounted on a children's stroller. The resolution, after Bayer filter, is 640x480.

Fig. 3.1 Example images from Caltech Dataset with annotations [99]. Pedestrians can be detected as a single person or as a group of people. The solid green boxes indicate full person or people detected. The yellow dashed boxes indicate the visible part of a pedestrian detected.

- TUD-Brussels dataset is more realistic for autonomous car applications because the images are captured from a driving car. The annotated pedestrians are not only in up-right but also in side-view position. The dataset annotates pedestrians whose sizes are higher than 48 pixels.

- Another popular dataset is from Daimler, presented in [101]. Its training set contains 15,560 (48x96) positive images and 6,744 person-free full images for extracting negative samples. The test set is a 27-minutes video of urban traffic taken from a vehicle. It contains more than 21,790 (640x480) images with 56,492 pedestrian (fully visible or partly occluded) labels. The minimum height of a label is 72 pixels.

- Caltech datset includes almost 10 hours of 640x480 30Hz video taken from street view in urban areas. It annotates 2,300 pedestrians with 350,000 bounding boxes. The annotations are grouped into three different scales: near (80 or more pixels), medium (from 30 to 80 pixels), and far (30 pixels or less). The medium scale is the most appropriate height to be detected because far pedestrians are too distant and it is too late to detect near scale ones.

- CityPersons is built based on Cityscapes dataset, which has instance-level labels for 30 visual classes including persons. The videos are captured from a stereo camera recording streets from 50 different cities. The total number of instance segmentation images is 5000. CityPersons also creates fine-grained category specifically for pedestrians to separate it from riders, sitting person, or others.
- EuroCity Persons is a new dataset targeting autonomous driving applications. It includes street scenes from 31 cities across 12 countries in Europe. Fifteen percent of the images are captured at night time.

Table 3.1 Popular pedestrian datasets

Dataset		INRIA	ETH	TUD-Brussels	Daimler	Caltech	CityPersons	EuroCity Persons
Year		2005	2007	2009	2009	2009	2017	2019
Training	Image size	64x128	640x480	720x576	48x96	640x480	2048x1024	1920x1024
	# pedestrians	1208	1578	1776	15.6k	192k	20k[d]	153k
	# positive images	1208	NA[a]	1092	NA	67k	NA	33k
	# negative images	1218	NA[a]	192	6.7k[c]	61k	NA	
Testing	Image size	64x128	640x480	640x480	640x480	640x480	2048x1024	1920x1024
	# pedestrians	566	9380	1326	56.5k	155k	11k	65k
	# positive images	566	NA[b]	508	21.8k	65k	NA	14k
	# negative images	453	NA[b]	NA	NA	56k	NA	

[a] The total number of positive and negative images are 490.
[b] The total number of positive and negative images are 1803.
[c] The resolution of these negative images is 640x480.
d The total number of annotation frames are 5000.

3.2 Evaluation methodology

Pedestrian detection algorithm results in detected bounding boxes which can be either true positives or false positives. Each detected bounding box has a confidence value, predicted by a classifier, which indicates the degree of certainty that a pedestrian is in the box. The confidence value must be greater than a given classification threshold.

There might be the case in which multiple detected boxes refer to a single pedestrian. One reason for this is the multi-scale detection. Non-maximal suppression (NMS) helps to mitigate this problem. The idea of NMS is to select the best candidate out of the nearby bounding boxes that potentially refer to a single pedestrian. The input of the algorithm is a list of bounding boxes, which actually is a list of coordinates, and a list of scores, e.g. confidence values, corresponding to those bounding boxes. The output of the algorithm is a subset list of the input bounding boxes since overlapping boxes are removed based on a given threshold. In other words, the algorithm effectively reduces the number of false positives. The pseudo-code for the algorithm is presented in Listing 3.1. The comments are added after the semicolon to make the code clearer.

```
Input:  BB = {bb_1, bb_2, ..., bb_N}, S = {s_1, s_2, ..., s_N}, T_NMS
        BB is the list of bounding boxes
        S is the list of the scores of the bounding boxes
        T_NMS is the overlap threshold
Output: BB_NMS , a subset of BB
begin
    BB_NMS = {}    ; BB_NMS is empty initially
    while ( BB is not ∅ ) do:
        m ← argmax  S        ;pick the index of the highest score
        M ← bb_m  ;put the bounding box with the highest score into M
        BB_NMS ← BB_NMS ∪ M ;put the bounding box into BB_NMS
        BB ← BB − M ;remove the bounding box from BB
        for bb_i in BB  do:
            if  (iou(M, bb_i) ≥ T_NMS) then
                BB ← BB − bb_i  ; remove the box that overlaps than threshold
                S ← S − s_i
            end
        end
    end
    return  BB_NMS, S
end
```

Listing 3.1 NMS pseudo-code

Having the list of detected boxes after NMS, IoU threshold is used to decide if each bounding box is a true positive (TP) or a false positive (FP). The common used IoU threshold is 50%, proposed by the PASCAL object detection challenges [106]. The IoU of a detected bounding box and a ground-truth is calculated as in Fig. 2.4. A detected box is a TP if its IoU with the ground truth is greater than the IoU threshold. Otherwise, the detected box is considered as a FP. A ground-truth that is not matched with any bounding box is counted as a false negative (FN).

Binary classifiers usually use the precision-recall curve to measure the performance as presented in Section 2.3.2. Pedestrian detection is also an application of binary classifier because the algorithm only needs to detect whether there is a pedestrian or not. However, since pedestrian detection for autonomous cars requires a low threshold of FPPI, it is preferable to evaluate its detection performance using the miss rate versus FPPI curve [99], [100].

Miss rate is calculated following Equation 3.1, based on a specific classification threshold; the higher threshold, the higher miss rate. Because a high threshold would create more FN and less FP.

FPPI is calculated following Equation 3.2. Higher classification threshold creates less false positives and thus smaller FPPI. In this work, I use the curve miss rate versus FFPI to indicate the performance of the algorithm. It is shown in Fig. 4.15. Instead, some works prefer to draw the curve recall versus FFPI [102], [103]. It is basically the same curve because miss rate $= 1 - recall$.

$$\text{miss rate} = \frac{FN}{TP + FN} \tag{3.1}$$

$$FFPI = \frac{\text{Total number of false positives}}{\text{Total number of images}} \tag{3.2}$$

Per-window detection does the classification task for fixed window-size images. In [100], the authors used window-size images of $64x128$ for the classification. The pedestrians' height in the testing dataset must be smaller than 128 pixels.

In contrast, per-image approach detects pedestrians on a full image, such as 640x480. It composes of locating the position and classifying if a pedestrian is presented in that position. In addition to classification task as in per-window approach, the detector also needs to (1) scale the image to detect small size (or far distance) pedestrians, (2) slide the detection window all over the scaled images to search for pedestrians, and (3) suppress the non-maximal bounding boxes to reduce false positives. Although the per-image approach contains more computational load, it is more practical for autonomous cars. In this work, I use the per-image approach.

Multiscale

Pedestrians can have different sizes depending on the distance from the camera as in 3.1. It is crucial to detect pedestrians from medium distance, corresponding to pedestrians' height from 30 to 80 pixels. Too close pedestrians (>80 pixels) are too late to be detected and less than 30 pixels pedestrians are too far. The distance between a pedestrian and the car can be inferred from the height of the pedestrian, measured in pixels, in the image as in Fig. 3.2. It can be approximately determined by Equation 3.3, in which f is the focal length of the camera, H is the height of the pedestrian, and h is the height in number of pixels of the pedestrian in the image.

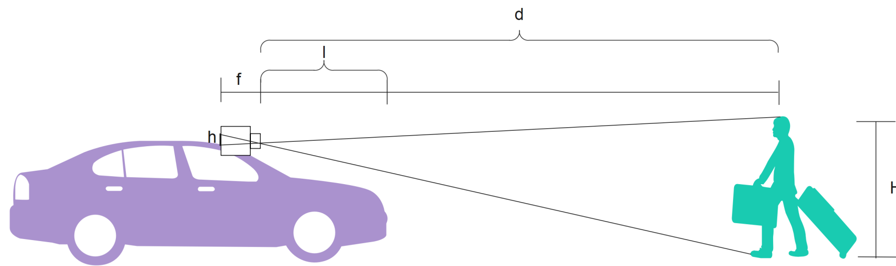


Fig. 3.2 Calculation the distance from the height of a pedestrian

$$\frac{h}{H} = \frac{f}{d} \tag{3.3}$$

According to the calculation in [107], assuming a pedestrian height is 1.8m, the height from 30 to 80 pixels corresponds to a distance from 20 to 60 meters. The algorithm should detect pedestrians at this distance before the car get closer. If the car runs at a speed of 55km/h (15m/s), it takes only 1.5s to get to the 80-pixel height pedestrian.

To detect at multiscales, each input image is scaled by a factor. Intuitively, input images need to be enlarged so that the algorithm can detect small size or far side pedestrians. Table 3.2 lists the sizes of scaled images with common scale factors. The input image size is 640x480 as in the Caltech and Daimler dataset. In [100], the window size is $64x128$. In the experiments, to reduce the scanning time, the size of the detection window is $96x160$. The height, 160, includes 32 pixels for top and bottom margins, as recommended in [100]. Therefore, the maximum height of a person can be detected is 128 pixels. Thus, to detect a 30-pixel height person, the input image should be scaled up approximately 4 times. From

Table 3.2, to achieve 4*x* higher resolution, with a scale factor of 1.2, the image needs to be scaled 7 times and the total number of scaled images is 8. If the scale factor is 1.1, the total number of scaled images would be 16. A smaller scale factor would give better detection performance with the cost of computing power due to the increasing number of scaled images.

Table 3.2 Scaled image sizes

Scale	Factor=1.05		Factor=1.1		Factor=1.2	
#	Width	Height	Width	Height	Width	Height
1	640	480	640	480	640	480
2	672	504	704	528	768	576
3	706	529	774	581	922	691
4	741	556	852	639	1106	829
5	778	583	937	703	1327	995
6	817	613	1031	773	1593	1194
7	858	643	1134	850	1911	1433
8	901	675	1247	935	2293	1720
9	946	709	1372	1029	2752	2064
10	993	745	1509	1132	3302	2477
11	1042	782	1660	1245	3963	2972
12	1095	821	1826	1369	4755	3566
13	1149	862	2009	1506	5706	4280
14	1207	905	2209	1657	6848	5136
15	1267	950	2430	1823	8217	6163
16	1331	998	2673	2005	9860	7395
17	1397	1048	2941	2206	11833	8874
18	1467	1100	3235	2426	14199	10649

Sliding window

The sliding window is the task that scans all over the image to detect for pedestrians. The stride between windows affects the detector's performance. Bigger stride would help to scan faster but it might hurt the detection accuracy. Figure 3.3 illustrates the sliding process of a 7*x*15 window over a 79*x*59 image. The Figure shows two detection windows drawn by dash lines with one pixel stride in the horizontal direction. Similarly, in the vertical direction, the stride is also one pixel.

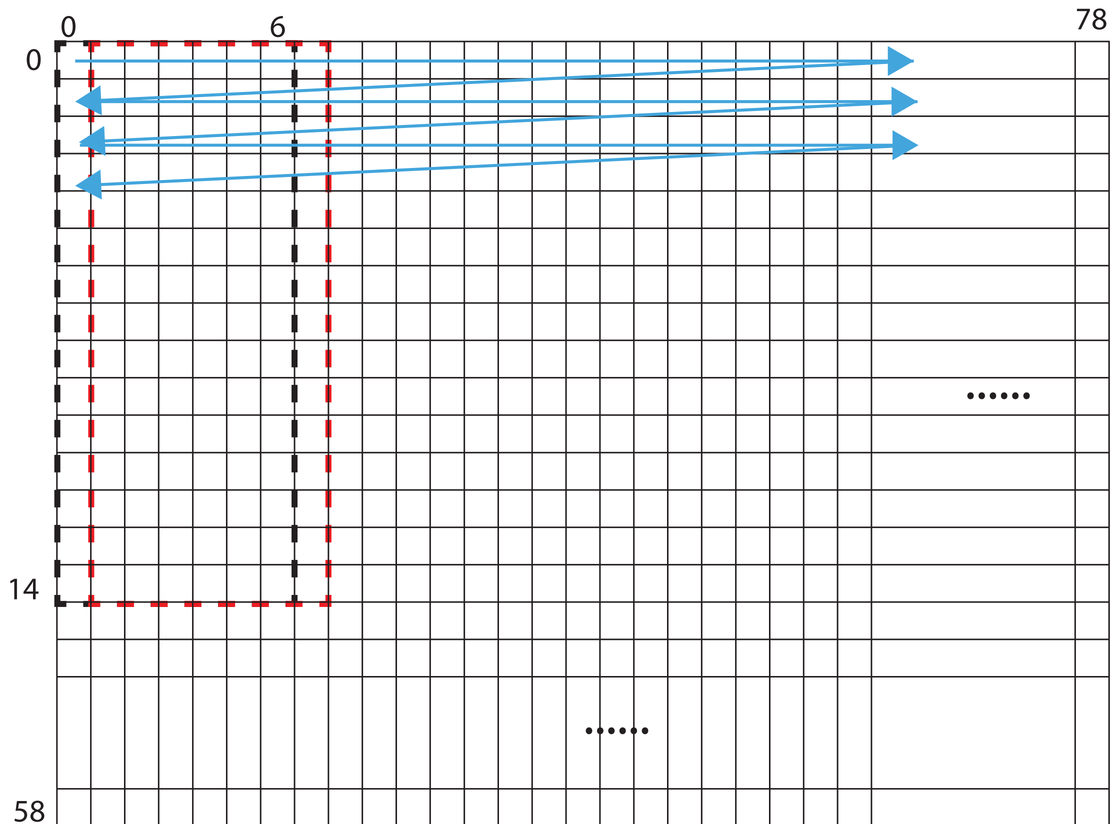


Fig. 3.3 Sliding a 7x15 window over a 79x59 image

3.3 State of the Art

Pedestrian detection methods were classified into two categories: hand-crafted features based approaches and deep features based approaches [108]. Popular hand-crafted features are SIFT [109], LBP [110], SURF [111], HOG [100], and Haar [112]. Handcrafted-features could be color, texture, or edge. A classifier will make predictions based on extracted features from input images. The most common used classifiers in these approaches are SVM and AdaBoost. The pipeline of hand-crafted features based approaches usually includes image scaling, feature extraction, classification, and non-maximal suppression. The block diagram for this approach is illustrated in Fig. 3.4.

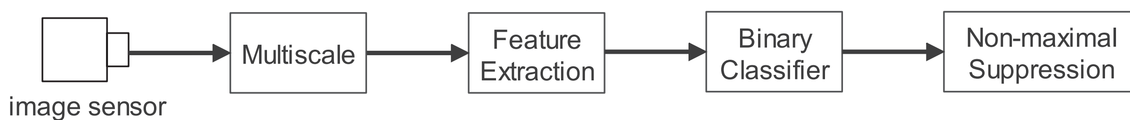


Fig. 3.4 Block diagram of a conventional pedestrian detection system.

The detector proposed in [113] is considered the first pedestrian detection system. It pioneered in combining motion feature with intensity feature to train the classifiers. The authors design a cascade system with multiple classifiers from simple to complex ones. The algorithm of the training process is Adaboost [114], [115]. The throughput of the system is 4 FPS for 360x240 frame size. However, the miss rate of the system is up to 95% on Caltech Pedestrian benchmark as shown in Fig. 3.5. This detection rate is measured at 10^{-1} FPPI and at reasonable setting on Caltech pedestrian dataset.

HOG+SVM, introduced in 2005 [100], achieved much higher accuracy on Caltech dataset. The approach uses HOG feature and SVM classifier. Its miss rate is 68% on Caltech dataset as in Fig. 3.5.

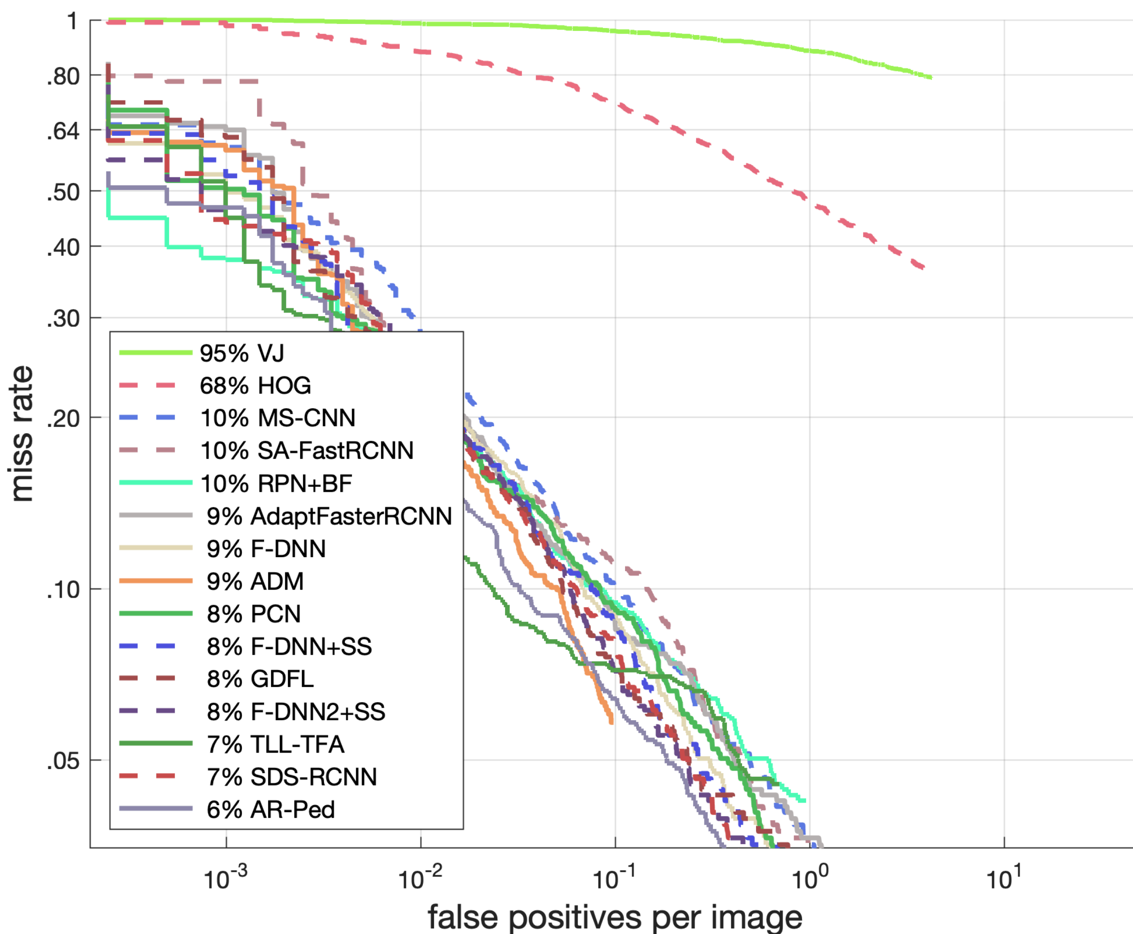


Fig. 3.5 Top detectors on Caltech Pedestrian detection benchmark [116]

.

Deep features based approaches are applying deep neural networks. In Fig. 3.5, top 13 detectors on Caltech pedestrian dataset are presented with the two baseline algorithms: VJ and HOG. The Figure is updated from the research result published in [99]. The top detectors,

starting from MS-CNN, have the accuracy of at least 10% miss rate. Table 3.3 lists out the classifiers for those detectors. All top algorithms are using deep neural networks. The block diagram of this approach for object detection is presented in Fig. 2.11.

AR-Ped achieves the best accuracy among these detectors. It has an accuracy of up to 6% miss rate and a throughput of 11 FPS, which is close to real time processing. The authors proposed autoregressive network, which aims to combine advantages of recurrent network (refining features) and ensemble network (diversifying features).

Deep neural network approaches usually have to deal with overfitting problem. It is the problem when the network performs very well on the training dataset but its accuracy degrades significantly on different and unseen dataset. The reason for this is the biasness towards the train/test dataset. Recently, Hasan et al. investigated this problem and proposed a generalizable pedestrian detector [117]. The authors prove that general object detectors can generalize better and thus improve accuracy on new datasets. Their model, based on general object detection network Cascade R-CNN, is trained on a merged dataset and achieved 2.5% miss rate on Caltech dataset.

Table 3.3 Performance results of detectors on Caltech dataset (the information on the Table is extracted from Caltech pedestrian dataset website [116]).

Algorithm	Year	Features	Classifier	Training	FPS
VJ [118]	2004	Haar	AdaBoost	INRIA	NA
HOG [100]	2005	HOG	Linear SVM	INRIA	NA
MS-CNN [119]	2016	pixels	deep net	Caltech+ImageNet	15
SA-FastRCNN [120]	2016	pixels	deep net	Caltech+ImageNet	1.7
RPN-BF [121]	2016	pixels	deep net+ AdaBoost	Caltech+ImageNet	2
AdaptFasterRCNN [104]	2017	pixels	deep net	Caltech+	NA
F-DNN [122]	2016	pixels	deep net	Caltech+	6
ADM [123]	2018	pixels	deep net	Caltech+ImageNet	NA
PCN [124]	2017	pixels	deep net	Caltech+ImageNet	NA
F-DNN+SS [122]	2016	pixels	deep net	Caltech+	0.4
GDFL [125]	2018	pixels	deep net	Caltech+	20
F-DNN2+SS [126]	2018	pixels	deep net	Caltech+	0.4
TLL-TFA [127]	2018	pixels	deep net	Caltech+	NA
SDS-RCNN [128]	2017	pixels	deep net	Caltech+ImageNet	4.8
AR-Ped [129]	2019	pixels	deep net	Caltech+ImageNet	11

Table 3.4 gives the SoA implementations on some popular pedestrian dataset for autonomous driving applications.

Table 3.4 Some popular datasets and the best implementations on them.

Dataset	Year	Best Result	Metric	Score	Speed (FPS)
Caltech [99]	2022	F2DNet [130]	Reasonable *MR*	2.2 %	7
CityPersons [104]	2019	CSP [131]	Reasonable *MR*	9.4 %	3
CityPersons [104]	2020	Pedestron [132]	Reasonable *MR*	7.5 %	NA
EuroCity Persons [105]	2020	Pedestron [132]	Reasonable *MR*	6.9 %	NA

3.4 HOG/SVM Pedestrian Detection

Even though it achieves high accuracy, DNN-based approach usually requires huge computing and memory resource. Its inference speed on CPUs/GPUs hardly achieves real-time processing. Porting DNN-based algorithms onto embedded platforms for power saving is a challenging task. The common strategy for porting is to trade-off accuracy for performance and resource constraints.

The conventional approach using a feature extractor and a binary classifier is more friendly to embedded platforms. This technique requires less computing resource and saves power consumption. HOG (Histogram of Gradients) feature [100] has proven to have good accuracy in human detection.

A typical block diagram for HOG+SVM pedestrian detection system is presented in Figure 3.6. A multiscale module is required so that the algorithm can detect pedestrians from different distance or sizes. Then, the HOG features of each scaled image are extracted and passed to the SVM classifier. Section 3.4.5 describes more detail about the SVM algorithm. NMS module helps to reduce false positives. Its pseudo-code is presented in Listing 3.1.

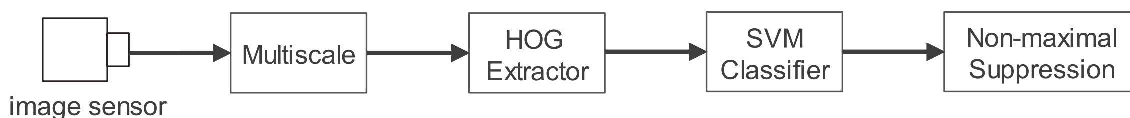


Fig. 3.6 Block diagram of a HOG+SVM pedestrian detection algorithm

A HOG feature extractor includes the processing steps shown in Fig. 3.7. The next sections describe the function of every module in this pipeline.

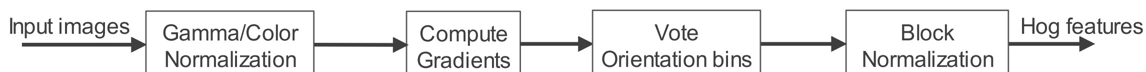


Fig. 3.7 HOG feature extractor block diagram

3.4.1 Gamma normalization

Gamma normalization is the first processing step of the chain. It helps to reduce the influence of illumination effects. The output of this block is the power law transformation of input pixels. Dalal et. al. has tried with log and square root functions [133]. The authors showed that square root transformation improves the detection performance for most of the object classes. Since photon noise in CCD sensors is proportional to the square root of intensity, taking square root of input pixels makes the effective noise approximately uniform. Therefore, the noise can be reduced after the gradient computation step. This transformation improves the performance, in the case of pedestrian detection, by 1% at 10^{-4} false positives per window (FPPW).

3.4.2 Gradient computation

Gradient computation is potentially affected by noise. A Gaussian filter can reduce the noise, but it also blurs the edges which are essential to later processing steps in the pipeline. Dalal et. al. [133] has shown that adding this filter will decrease the detector's performance. Therefore, the implementation of the HOG algorithm can exclude this step to save computing resources.

To compute image gradients, a mask is slid and convoluted over all the image. An ideal mask is the one that provides good detection performance. Table 3.5 is derived from [133]. The simple mask 1-D centred is simple and gives the best performance of 11% miss rate at 10^{-4} FPPW.

Table 3.5 Different masks for gradient computation

Mask type	1-D centred	1-D uncentred	1-D cubic-corrected	2x2 diagonal	3x3 sobel
Operator	$[-1,0,1]$	$[-1,1]$	$[1,-8,0,8,-1]$	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$
				$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$
Miss rate	11%	12.5%	12%	12.5%	14%

Applying 1-D centred mask, gradients of a pixel (x,y) are calculated in horizontal and vertical directions following Equation 3.4 and 3.5 respectively.

$$G_x(x,y) = I(x+1,y) - I(x-1,y) \tag{3.4}$$

$$G_y(x,y) = I(x,y+1) - I(x,y-1) \tag{3.5}$$

Then, the magnitude and the orientation gradient at pixel (x,y) are computed by Equation 3.6 and 3.7.

$$G(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2} \tag{3.6}$$

$$\phi(x,y) = \arctan \frac{G_y(x,y)}{G_x(x,y)} \tag{3.7}$$

The output of the Gradient computation stage is the magnitude and orientation gradients of all input pixels.

3.4.3 Orientation bin voting

HOG feature is extracted over local spatial regions called cell. Typically, a cell has a size of 8x8 pixels. Each cell consists of 64 pairs of magnitude and orientation gradients. Depending on its associated orientations, magnitude gradients are accumulated to the corresponding bins. A cell histogram with nine bins is illustrated in Fig. 3.8. Figure 3.8b describes in detail how the orientation of the gradient is voted into a range of 9 bins using the scale from 0 to 180°. The magnitude G, in this example, should be accumulated to bin 2 because its orientation is approximately 30°. This simple method, voting the whole magnitude to the nearest orientation bin, would create aliasing effects.

A more accurate way to vote is linear interpolation, which accumulates the magnitude to two neighbour orientation bins. The percent of magnitude voted to each bin depends on the distance from the orientation gradient to the center of the bin. For example, if the center of the neighbour bins are $bin1$ and $bin2$, there corresponding weights, named $w1$ and $w2$, are calculated as in Equation 3.8.

$$\begin{aligned} w1 &= \frac{bin2 - \phi(x,y)}{bin2 - bin1} \\ w2 &= \frac{\phi(x,y) - bin1}{bin2 - bin1} \end{aligned} \tag{3.8}$$

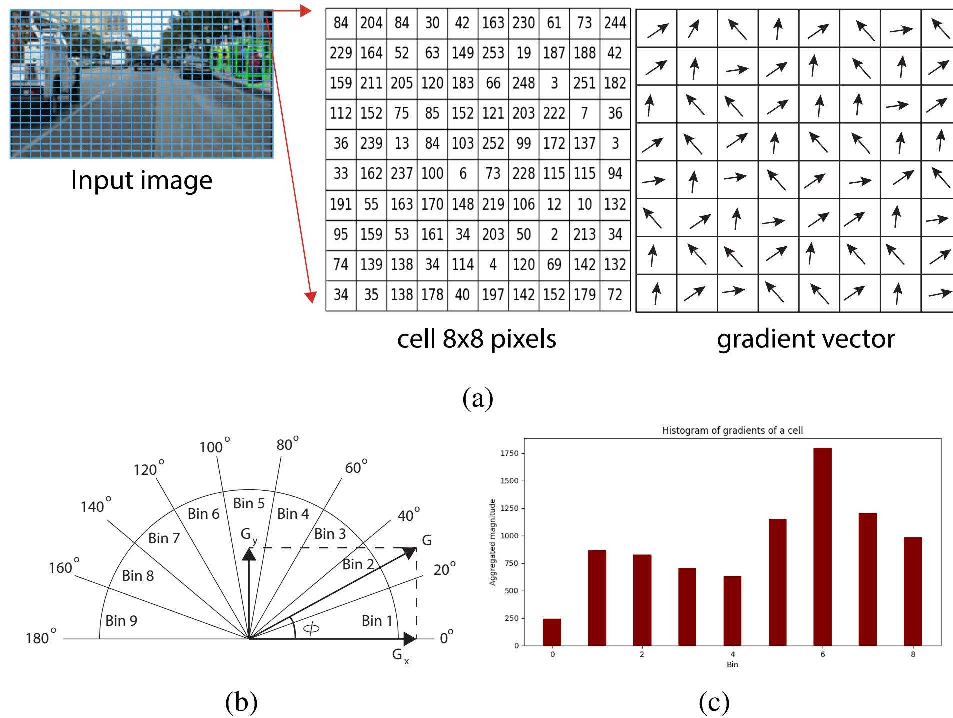


Fig. 3.8 An illustration of how HOG features are generated. a) Image is divided in 8x8 cells. The pixels' value of a cell are randomly created to plot the histogram in the subfigure 3.8c. The gradient vector is also illustrated; each cell has $8x8$ pixels. b) A pixel has a magnitude gradient G, and orientation gradient ϕ ranged from 0 to 180°; these gradients are calculated from G_x and G_y. c) All 64 pixels' values in subfigure 3.8a vote their magnitude gradient to the appropriate orientation bin among 9 bins to create the cell 's vector.

The magnitude gradients of the pixel at (x, y) to be accumulated to bin1 and bin2 are $w1 \times G(x, y)$ and $w2 \times G(x, y)$ respectively. The final HOG feature of a cell is a 9-dimensional vector, corresponding to 9 bins. Reducing the number of bins would decrease the detection performance while increasing it makes no significant improvement [133].

3.4.4 Block Normalization

Block normalization is an important step, it helps to improve the detection performance. It aims at normalizing contrast caused by illumination difference. There are different schemes of normalization. The two best schemes are L1 square root and L2; they provide equal performance for person detection.

Block is a spatial region which encompasses proximity cells. A block typically has a size of 2x2 cells, which equals to 16x16 pixels. Overlapping 75% and 50% between blocks improves the detection performance by 4% and 2% respectively [133]. The L1 square root

and L2 normalization can be done following the equations in 3.9 and 3.10. In the equation, the v at the left of the arrow is the cell HOG features after normalizing. The v at the right side of the arrow is the features before normalizing. L1 and L2 norm in those equations are calculated for the whole block.

$$v \leftarrow \sqrt{\frac{v}{\|v\|_1 + \varepsilon}} \tag{3.9}$$

$$v \leftarrow \frac{v}{\sqrt{\|v\|_2^2 + \varepsilon^2}} \tag{3.10}$$

The output of the normalization stage is the final HOG feature. Concatenating all the HOG features of all blocks provides the final HOG feature of the image. To do detection, input images are divided into spatial local areas named detection windows. Figure 3.9 describes the detection window, the block, and the cell with respect to the image to be detected.

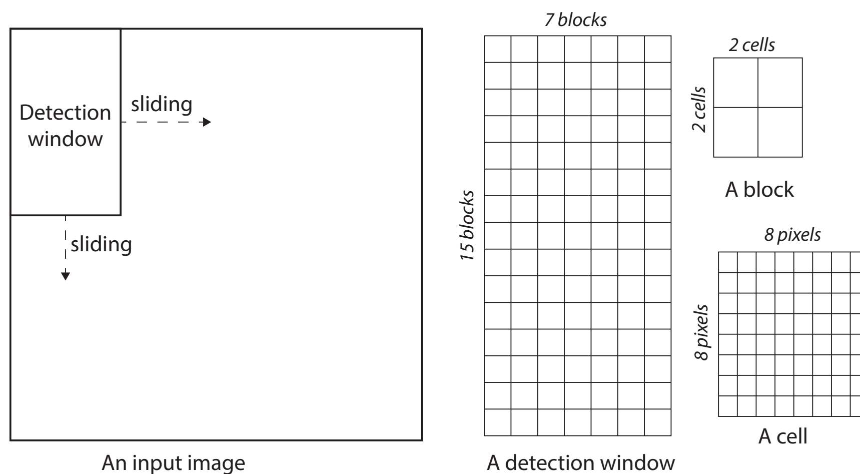


Fig. 3.9 Illustration of cell, block, and detection window in an image. The detection window will slide all over the image to detect object of interest. Each window has a size of 7x15 blocks, each block is 2x2 cell, and each cell is 8x8 pixel.

3.4.5 Support Vector Machine

SVM is known as Support Vector Networks [134]; it is a type of machine learning. SVM bases on supervised learning model, the model learns from a set of labelled training data. After learning, the model can predict the correct label for previously unseen data. It is usually used as a binary classifier in pedestrian detection system (Fig. 3.6).

If labelled training data pairs (x_i, y_i) are given, where $i = 1, ..., l$ and l is the number of pairs, $x_i \in R^n$, and $y_i \in \{-1, 1\}$, SVM model is obtained by solving the optimization problem which minimizes the expression 3.11 with the constraint in expression 3.12.

$$\frac{1}{2}W^T W + C\sum_{i=1}^{l} \xi_i \tag{3.11}$$

$$\begin{cases} y_i(W^T x_i + b) \geq 1 - \xi_i, \\ \xi_i \geq 0 \end{cases} \tag{3.12}$$

The three variables to be found in this problem are: W, b, and ξ. The solution to this optimization problem is illustrated in Figure 3.10. The problem is reduced into 2 dimensions for easy explanation. The key idea is to find a line that separates the data points from the two different classes, shown in positive symbol and negative symbol. The line should not only separate all training data points correctly but also create a bigger margin because it will help to classify testing data better. Constant C, in 3.11, is given by user. It is used to trade-off between a bigger margin and a smaller number of training data points that fall into the wrong side of the line. For instance, data points $\vec{x}_1, \vec{x}_2, and \vec{x}_3$ are separated wrongly in Figure 3.10. The variables ξ_1, ξ_2, and ξ_3 represent how big the errors are.

The second operand in 3.11 expresses the penalty of training errors to obtain a maximum margin in the first operand. If C is big enough, the separating line, if it is found, completely classifies training data points, without any training error. The training data, in this case, is linear separable.

If the training data is non-separable, the separating line is allowed to have some wrong classified training data points. This solution is called *soft margin hyperplane*. Nevertheless, it would be good to have a separable training data. Therefore, function ϕ in 3.13 is introduced to transform the training data $\vec{x}_i$ into a higher dimensional space that would potentially make it linear separable.

$$\begin{cases} y_i(W^T \phi(x_i) + b) \geq 1 - \xi_i, \\ \xi_i \geq 0 \end{cases} \tag{3.13}$$

What we have seen is the graphical illustration of the soft margin hyperplane problem. To solve it, one way is to use the Lagrange dual problem. The problem becomes finding the

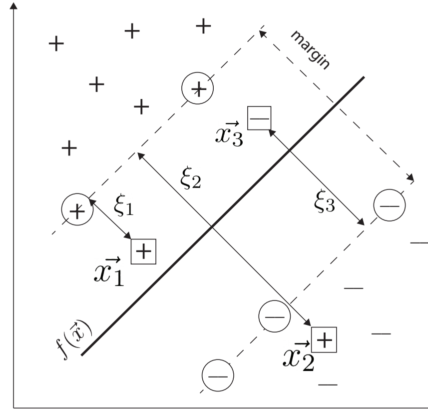


Fig. 3.10 Illustration of 1) Circled data points which are also called support vectors, 2) The bold line $f(\vec{x})$ which separates the two types of data points: positive symbol and negative symbol, 3) Data points in rectangular shape which are on the wrong side of the bold line, 4) Data points belong to two classes, represented by positive symbols and negative symbols, are separated correctly, 5) The margin corresponding to the bold line that separates data points. SVM selects the bold line so that the margin is biggest while the number of rectangular points is the smallest.

maximum of the expression in 3.14 subject to 3.15.

$$\sum_{n=1}^{l} \lambda_n - \frac{1}{2} \sum_{n=1}^{l} \sum_{m=1}^{l} \lambda_n \lambda_m y_n y_m \phi(x_n)^T \phi(x_m) \tag{3.14}$$

$$\begin{cases} \sum_{n=1}^{l} \lambda_n y_n = 0 \\ 0 \leq \lambda_n \leq C, \forall n = 1, 2, ..., l \end{cases} \tag{3.15}$$

Solving problem 3.14 gives us λ, which is Lagrange multiplier. It is used to calculate W and b. It is proven that w and b can be rewritten as in Equation 3.16 and 3.17, where M and S are the set of n and m index, $M = \{n : 0 < \lambda_n < C\}$ and $S = \{m : 0 < \lambda_m \leq C\}$.

$$w = \sum_{m \in S} \lambda_m y_m \phi(x_m) \tag{3.16}$$

$$b = \frac{1}{N_M} \sum_{n \in M} (y_n - W^T \phi(x_n)) = \frac{1}{N_M} \sum_{n \in M} (y_n - \sum_{m \in S} \lambda_m y_m \phi(x_m)^T \phi(x_n)) \tag{3.17}$$

Having W and b, the prediction is done by evaluating the expression 3.18.

$$w^T\phi(x)+b=\sum_{m\in S}\lambda_m y_m\phi(x_m)^T\phi(x)+\frac{1}{N_M}\sum_{n\in M}(y_n-\sum_{m\in S}\lambda_m y_m\phi(x_m)^T\phi(x_n)) \quad (3.18)$$

However, it is costly to compute $\phi(x)$ because it has a big number of dimensions. The kernel trick allows us not to evaluate $\phi(x)$ for every data x. Instead, it is only necessary to calculate $\phi(x)^T\phi(z)$ based on any x and z in the data set.

Defining kernel function $k(x,z)=\phi(x)^T\phi(z)$, expression 3.18 is rewritten as in 3.19.

$$w^T\phi(x)+b=\sum_{m\in S}\lambda_m y_m k(x_m,x)+\frac{1}{N_M}\sum_{n\in M}(y_n-\sum_{m\in S}\lambda_m y_m k(x_m,x_n) \quad (3.19)$$

The four popular kernels are:

- linear: $K(x_i,x_j)=x_i^T x_j$.
- polinomial: $K(x_i,x_j)=(\gamma x_i^T x_j+r)^d, \gamma>0$.
- radial basic function (RBF): $K(x_i,x_j)=exp(-\gamma\|x_i-x_j\|^2), \gamma>0$.
- sigmoid: $K(x_i,x_j)=tanh(\gamma x_i^T x_j+r)$.

Where γ, r, d are kernel parameters.

In this thesis, SVM is used to train and classify pedestrians in images. The process consists of two phases: training and classifying. Because of the high computational complexity, the training phase is not usually used in a real-time embedded system. Instead, it is done offline and generates a model. The classification phase will then use that model to infer predictions.

Training phase

The input of a training phase is training data, which, in this case, is the set of HOG feature vector of every training image and their corresponding labels. Every training image includes a fixed number of detection windows, which depends on the size of the image and the detection window. A label, corresponding to a detection window, indicates either a pedestrian or non-pedestrian. It depends on whether a pedestrian is present in the detection window or not. Mathematically, training is the process of solving the Equation 3.20 [135], where $\vec{x}_i$ is the HOG feature vector of the window number *ith* of the input training image, y_i is its corresponding label which could be either 1 or -1, and N is the number of input training image including both positive and negative samples.

$$y_i(\vec{W}\cdot\vec{x}_i+b)\geq 0, i=1,...,N \quad (3.20)$$

By solving equation 3.20, vector $\vec{W}$ and b are determined. In this thesis, open source LIBSVM with linear kernel [136] is used for the training phase and thus obtaining $\vec{W}$ and b.

Classification phase

The classification phase uses the trained model to infer prediction. The confidence value $y(\vec{x})$ of a detection window, which will be compared to a threshold to determine an object as a pedestrian or not, is calculated from Equation 3.21. Weight vector $\vec{W}$ and the bias b are provided by the model while $\vec{x}$ is the HOG feature vector of the window that needs to be detected.

$$y(\vec{x}) = \vec{W}^T \cdot \vec{x} + b \tag{3.21}$$

3.5 Summary

This chapter focuses on the pedestrian detection problem. It defines the scope of the problem under investigation. Different databases are available for developing and evaluating pedestrian detection systems. In this thesis, Caltech database is selected because its data is real street scene and from a viewpoint of a car. Evaluation methodology might also be different among implementations. In this thesis, the FPPI method is used to measure the detection accuracy. This method is more complicated and practical for autonomous cars. The detection window needs to slide all over the input images. The sliding method potentially creates false positives. Besides, to detect pedestrians at different distance using a fixed detection window size, input images are scaled up at different steps. This is another source that creates false positives. Therefore, a NMS algorithm is in place to filter out false positives.

The mainstream approaches to tackle the problem are applying deep neural networks. Top accurate detectors on Caltech dataset are based on these approaches. Nevertheless, deep neural networks usually require huge computing and memory resource. Their inference speed hardly achieves real-time processing on CPUs/GPUs.

Conventional approaches are more friendly to embedded platforms which are usually resource-constrained and consume less power. These approaches are based on handcrafted-features and binary classifiers.

In this thesis, I choose the second approach and implement the system on FPGAs to see the trade-off between processing speed and detection accuracy. Specifically, HOG is selected as the feature for detection and SVM is the classifier. In this chapter, HOG and SVM algorithm are investigated so that they will be implemented in hardware in the next chapters.

Chapter 4

HOG/SVM pedestrian detection implementation

4.1 System architecture

This chapter details the micro-architecture of the designed RTL-based pedestrian detection system. The system targets, including a HOG feature extractor and an SVM classifier, are high throughput and energy efficiency. System architecture is depicted in Fig. 4.1 and it is mapped onto the low-cost educational DE1-Soc development board [137]. The system includes hardware accelerators mapped into the programmable logic (PL) part and software running on the Hard Processor System (HPS) part. The input to the system are pixels from D5M image sensor kit [138].

To achieve real-time performance, the critical paths (HoG and SVM highlighted as custom in the system) are accelerated on FPGA programmable logic part of the Cyclone V SoC chip. The micro-architecture detail of these two modules is presented in section 4.2 and 4.3.

Other functions such as input images display and drawing bounding boxes are more software friendly and are running on ARM processor integrated in the Cyclone V SoC.

Figure 4.1 shows that the HOG+SVM design does not access external DDR3 memory at any pipeline stage. Accessing external memory requires higher latency and energy consumption than using internal/on-chip memory. This implementation only uses external memory to store input images pixels from the sensor and positions of bounding boxes surrounding predicted pedestrians.

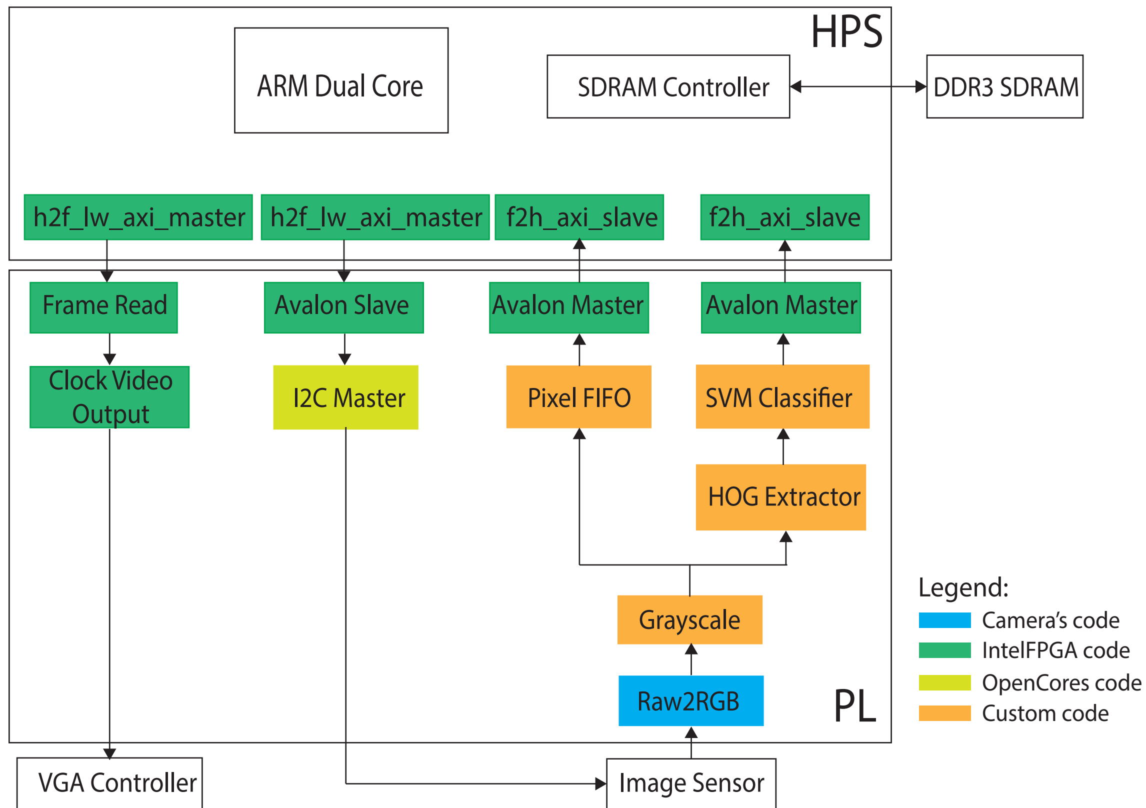


Fig. 4.1 System diagram of the RTL-based pedestrian detection system on DE1-SoC.

The camera settings can be configured by the software code through I2C interface. For that purpose, an open source I2C master IP core is instantiated in PL to communicate with the I2C slave in the image sensor.

The image sensor interface requires some Verilog code for capturing raw pixels (Bayer pattern) and generating RGB pixels. Grayscale pixels are then created from the R, G, and B pixel values by the RAW2RGB module in Fig. 4.1. Table 4.1 shows the resolution of the images while traversing through different modules.

With the settings detailed in Appendix A, I restrict the image size to 1280x960 pixels at the output of the image sensor. The RAW2RGB module uses Bayer pattern to generate RGB images which are 4x smaller. Therefore, RGB images are at VGA size (640x480).

The Bayer pattern is illustrated in Fig. 4.2. Every group of four raw pixels includes two green pixels, one red pixel, and one blue pixel. Each group will generate one RGB pixel, in which the green component is the average of the two green raw pixels.

Finally, the Grayscale module generates grayscale images to be fed into the HOG extractor engine. This module only converts pixels and keeps the image resolution unchanged.

It is worth noting that the frame rate at the output of the image sensor is only 11 FPS as calculated in Appendix A according to the sensor specification provided by the producer. The maximum frame rate of the sensor can be up to 150 FPS at VGA resolution and 96 MHz pixel clock. For capturing and visualizing real input images from the camera and prediction results, the frame rate of 11 FPS is selected.

Grayscale images are buffered in a FIFO to be sent to the external memory for visualizing. The external memory also stores prediction results from the SVM Classifier. Particularly, the coordinates of the bounding boxes of every input image are stored. Consequentially, the images from the camera, together with the prediction bounding boxes (if any), are visualized on a VGA monitor through the VGA Controller module.

The green boxes in Fig. 4.1 (provided by the chip maker) are: (i) the Avalon master bus interface wrapper to send pixels and prediction results to the HPS memory; (ii) Avalon slave bus interface to write register configurations for the I2C Master module; and (iii) Avalon slave bus interface to write pixel data from the HPS to the Frame Read module and later to the display through the VGA controller. All these boxes are available in the library IP provided by the chip maker.

Table 4.1 Image resolution from the camera to the HOG Extractor

Module	Resolution
Max. sensor resolution	2560x1920
Max. RAW2RGB image size	1280x960
RAW2RGB Bayer image size	640x480
Grayscale image size	640x480

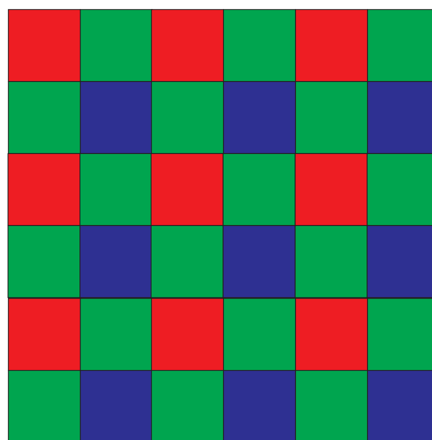


Fig. 4.2 Bayer pattern illustration.

In conclusion, the key parts of the pedestrian detection system are the HOG Extractor and the SVM Classifier modules. These are latency critical paths of the system that need HW acceleration for real time detection. The input images to the HOG+SVM engine are 640x480 grayscale images with a frame rate of 11 FPS. The next two sections will present the detail implementation of HOG and SVM modules.

4.2 HOG extractor pipeline design

4.2.1 The pipeline design

HOG algorithm is already presented in section 3.4. Here, an RTL implementation of the algorithm on FPGA is proposed. The detailed architecture of the HOG pipeline is shown in Figure 4.3.

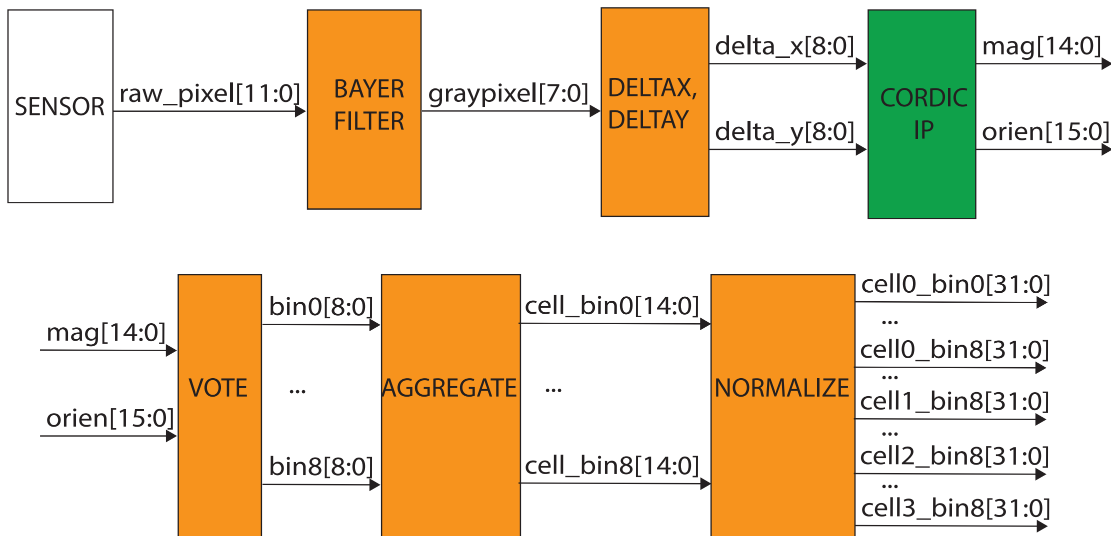


Fig. 4.3 HOG extractor block diagram

The pipeline starts with the Bayer Filter which converts raw pixels into RGB pixels and grayscale pixels. Grayscale image is based on component Y of the YUV color space. This component is calculated from RGB components as in Equation 4.1. The equation is implemented approximately using logic shift operations as in Equation 4.2. Finally it is implemented in hardware to generate grayscale images.

$$Y = 0.299 \times R + 0.587 \times G + 0.114 \times B \tag{4.1}$$

$$Y = (R >> 2) + (R >> 5) + (G >> 1) + (G >> 4) + (B >> 4) + (B >> 5) \quad (4.2)$$

The next stage in the pipeline computes the intensity difference of neighbor pixels as in Equation 3.4 and 3.5 for horizontal and vertical directions respectively. The objective of the pipeline is to process every input pixel effectively so that pixels do not need to be stored in any external memory and do not block the following pixels. This also means that helps reduce both total latency and power consumption. It also means that the throughput of the pipeline would be higher if pixels are fed in faster. To do that, line buffers are used to temporarily store pixels as shown in Fig 4.4.

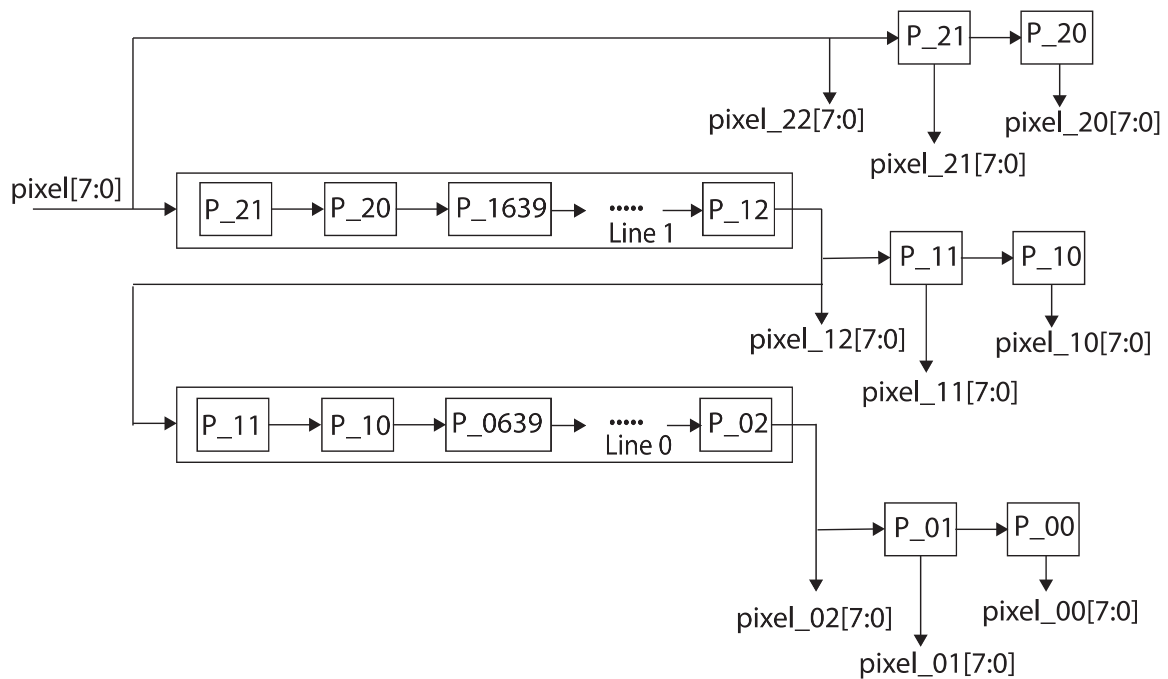


Fig. 4.4 Pixel line buffers

The length of the line buffer is double the width of input images. In Fig. 4.4, the line buffer includes line 0 and line 1, each stores 640x8 bits. The output of the line buffer, as illustrated in Fig. 4.4, are at position P_12 and P_02, which refer to the second pixel of row 1 and row 0 of input images respectively. At every pixel clock, a new pixel is shifted into the line buffer and, thus, all the pixels in the buffer are also shifted with new values (in Fig. 4.4, pixels are shifted to the right).

Line buffer helps to effectively store the required window on a memory size of two rows. This design is scalable because its size only depends on the row's size of the image.

More importantly, there is no latency cost in accessing pixels' values. The throughput of the pipeline only depends on the clock that feeds pixels in.

At the right side of the line buffer, there is a window buffer with six 8-bit registers. These registers together with the outputs from the line buffer and the newly input pixel value form the 3x3 window used to implement Equation 3.4 and 3.5. For instance, according to Fig. 4.4, *delta_x* of pixel P_{11} (pixel 1 in row 1) is calculated using P_{10} and P_{12} as in Equation 4.3. Similarly, *delta_y* is calculated using P_{01} and P_{21} as in Equation 4.4. Delta values can be negative numbers, so i use nine bits to represent signed integers from -256 to 255.

$$delta_x(1,1) = P_{10} - P_{12} \tag{4.3}$$

$$delta_y(1,1) = P_{01} - P_{21} \tag{4.4}$$

The CORDIC IP in Intel-Altera FPGA library is used to compute the magnitude (signals gra[14:0]) and the orientation (signals orien[15:0]) of the gradients. The specific algorithm for this task is vector translate, which converts an input vector defined by its x and y coordinates into magnitude and angle of the vector. This algorithm essentially implements Equations 3.6 and 3.7 with the difference that $atan2$ function is done instead of $atan$. At this stage, fixed-point representations are required to represent results since integer numbers would end in low precision results. The functional settings for the IP are shown in Fig. 4.5. Input data width is set to be sixteen, in which there are seven fractional bits and a sign bit. Therefore, the delta values, which is nine-bit width, need to be appended with 7 fractional bits.

For the output configuration, the fraction part of angle output value is configured to 13 bits, which has an accuracy of up to 0.000061. Based on this selection, all other settings are derived automatically by the IP. In particular, the fraction part of magnitude output is 6-bit depth, corresponding to an accuracy of 0.008. This accuracy is acceptable since magnitude output is in the range from 0 to 360.

CORDIC IP has also parameters related to speed performance as shown in Fig. 4.6. The target working frequency is set to 100 MHz although it can be set higher because the frequency of the whole pipeline is 50 Mhz (the pixel clock).

The next stage of the pipeline is the voting module. Its input are the outputs from CORDIC IP: magnitude and orientation gradient. The magnitude gradient is an unsigned 15-bit fixed point number, and the orientation gradient is a 16-bit signed fixed-point number. Depending on the orientation gradient, the magnitude gradient of each pixel will be voted to appropriate bins. In the implementation of this thesis, there are 9 bins covering the range from 0 to π as illustrated in Fig. 3.8b. Since the range of the orientation gradient is $[-\pi, \pi]$,

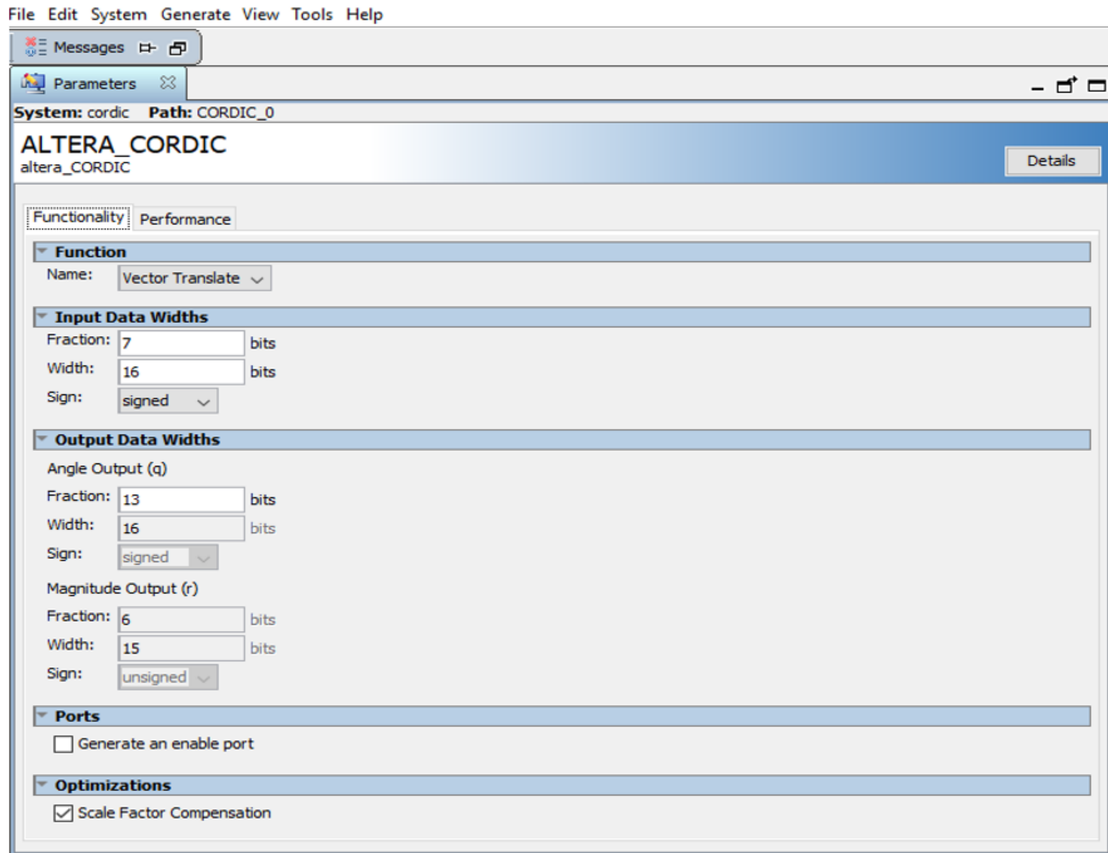


Fig. 4.5 CORDIC IP functionality settings

negative orientation input will be added an amount of π to make it in the range of the nine bins.

VOTE module uses bilinear interpolation method to vote a magnitude gradient to appropriate bins. The magnitude will be distributed to two bins nearest to its orientation gradient value. Bilinear interpolation is done with 16-bit fixed point numbers and final output bins are rounded to be 9-bit unsigned integers. This technique is described in detail in section 3.4.3.

The 9-bin output from VOTE module is the value calculated for every pixel. The pipeline shifts one pixel in every clock cycle since the horizontal blank setting is zero. Therefore, VOTE module provides 9-bin output at every clock cycle. This set of 9 bins is the histogram of a pixel. Each bin is a magnitude gradient, represented by 9-bit unsigned integers. Fractional part can be truncated because it is very small compared to the integer part.

Next, the AGGREGATE module accumulates 64 histograms of 64 pixels of a cell to obtain cell histogram. In this implementation, the cell size is 8*x*8 as in HOG's original paper [100]. Figure 4.7 illustrates an 8*x*8 cell in a 640*x*480 image.

Fig. 4.6 CORDIC IP performance setting

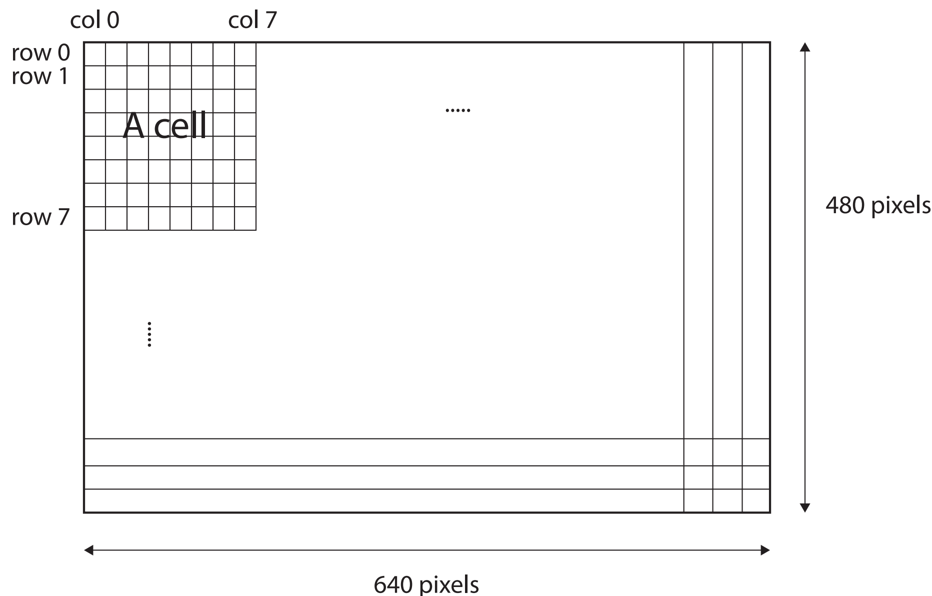


Fig. 4.7 Illustration of an 8*x*8 cell in a 640*x*480 image

With a 640*x*480 image size, the total number of cells in an image is 80*x*60 and HOG feature is a vector of 80*x*60*x*9 dimension since each cell is represented by 9 bins. Because pixels are streaming in sequentially, the AGGREGATE module cannot have all gradient information from 64 pixels at one time. As shown in Fig. 4.9, after having 8 pixels of row 0 of cell 0, the module needs to wait for other pixels of row 0 of the image to come in before the pixels of row 1. Therefore, the module can only generates a partial HOG vector for cell 0. This partial vector aggregates 8 pixels in a row of a cell. A cell needs to accumulate a total of 8 partial vectors. Similarly, the next 8 pixels will form the partial vector for cell 1 and so on. There are 80 cells in a row. Hence, a line buffer with 80 elements is created to store these partial vectors. The logic design for the line buffer is presented in Fig. 4.8.

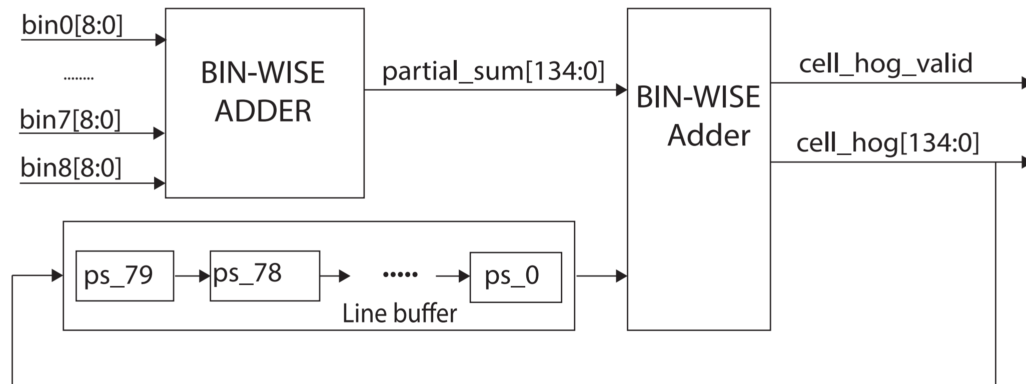


Fig. 4.8 AGGREGATE module block diagram

To generate the full HOG feature of a cell, it is necessary to aggregate 8 partial cell vectors from 8 rows. The name of the partial vector of a row is $partial_sum[134:0]$ and it is the output of the left Bin-Wise Adder of Fig. 4.8. The width of this result is 135 bits, representing 9 bins, 15-bit width each. The reason why each bin needs 15 bits is that the maximum value of a magnitude gradient is 360 and if all 64 pixels vote to only one bin, that bin would take the value of 64 times 360, which is $23,040$. The Bin-Wise Adder on the right of Fig. 4.8 is used to add the accumulated sum of a cell in the line buffer and a new partial sum of a new row of the same cell. When the final row of a cell is generated and added to the accumulated sum stored in the buffer, the $cell_hog_valid$ signal is set and the $cell_hog[134:0]$ output becomes the hog vector of the cell. This value goes into the next stage of the pipeline: normalization. The line buffer will not store this value and it will clear that buffer to zero to reuse it for another cell. This is an optimized way of using the on-chip memory resource. Only a line buffer of 80 elements is used for buffering and calculating up to 80x60 cells' hog vector. Besides, with this line buffer, the full feature vector of the next cell is available 8 cycles later.

Finally, cell features are block-wise contrast normalized. In this design, each block has four cells and L2 normalization [100] is chosen for the sake of accuracy and simplicity. Fig. 4.9 shows the first two blocks of a 640x480 input image. Block 0 includes the four cells: cell 0, cell 1, cell 80, and cell 81. Block 1 shares cell 1 and cell 81 with block 0.

At this normalization stage, the four cells' vectors are required for every block so that it can evaluate Equation 4.5. In the equation, v_c is the cell hog features of a cell, $||v||_2$ is the L2-normalization of the four cell hog features in the block, and v_{cL2} is the normalized feature of a cell in the block. A small constant, ε, is added to avoid dividing by zero. In this design, it is set as 1 for simplicity.

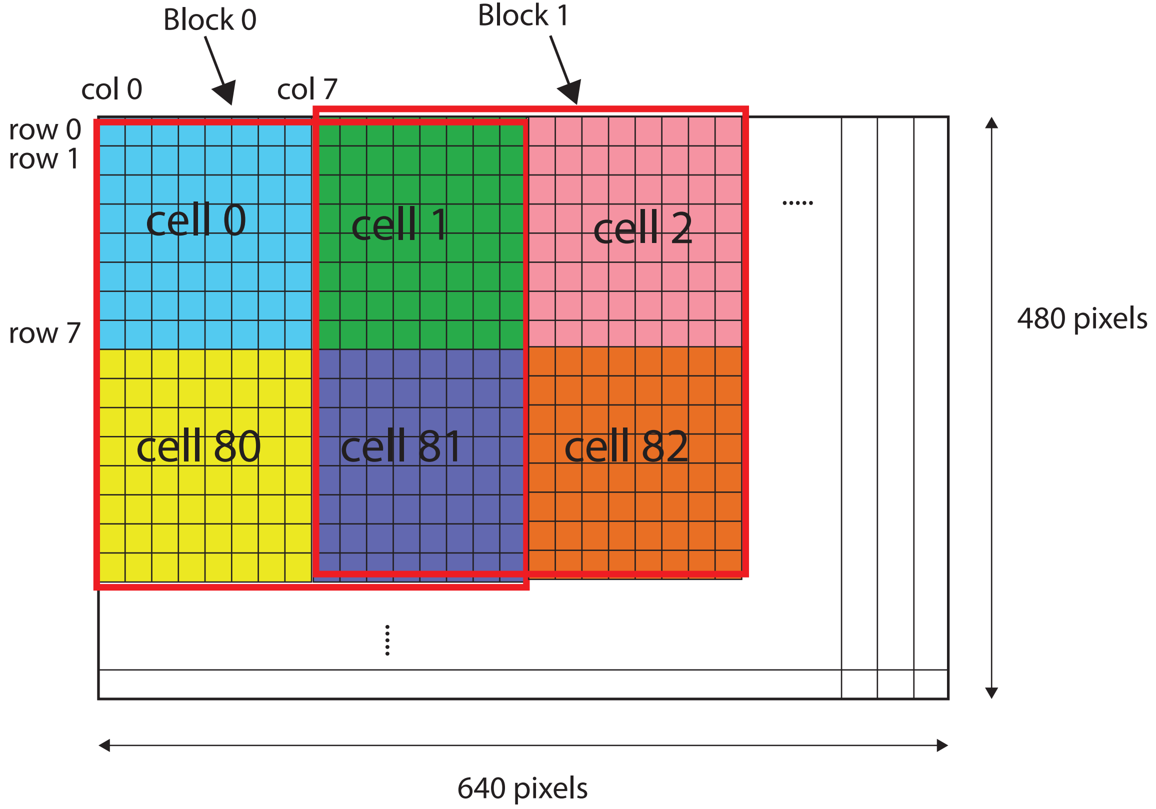


Fig. 4.9 Blocks and cells in an 640x480 image

The calculation of L2-normalization is presented in Equation 4.6, in which v_c represents feature value at cell c. The sum is calculated bin-wise. Therefore, the result of Equation 4.6 is a vector of 9 bins.

The numerator of Eq. 4.5 is the hog vector of a cell, including 9 bins. Therefore, to implement Eq. 4.5 in hardware, it is required to have 9 different dividers.

$$v_{cL2} = \frac{v_c}{\sqrt{\|v\|_2^2 + \varepsilon^2}} \tag{4.5}$$

$$\|v\|_2 = \sqrt{\sum_{c=0}^{c=3} v_c^2} \tag{4.6}$$

From Equation 4.5 and 4.6, the final normalized HOG is calculated in Equation 4.7.

$$v_{cL2} = \frac{v_c}{\sqrt{\sum_{c=0}^{c=3} v_c^2 + 1}} \tag{4.7}$$

The pipeline for implementing Eq. 4.7 is shown in Fig. 4.11. The input to the pipeline are the thirty-six bins of the four cells. A valid signal ($four_cell_valid$) is required to indicate when the bins' values are available. This signal is controlled by a finite state machine illustrated in Fig. 4.10. The state machine collects four valid cell vectors, which are the outputs from the AGGREGATE stage. For example, the four cells of block 0 are cell 0, cell 1, cell 80, and cell 81. It is worth noting that they are not coming at a time. Cell 80 needs to wait until the first pixels of the rows from seventh to fifteenth to be available.

At first, the machine is in IDLE state. It changes to WAIT state when it detects the first valid cell vector and it stays eighty valid cell vectors in this state. As illustrated in Fig. 4.9, these are the vectors from 0 to 79. When the state detects the eightieth valid cell vector, it goes into NORM state. At this state, every cell vector valid comes would lead to a valid block, meaning $four_cells_valid$ signal is set to logic one. It is worth noting that a block encompasses four cells which locate at two different cell rows (means sixteen different pixel rows). For the first row of blocks, it is expected to have 79 valid blocks. Then, the FSM goes to the SKIP state, that is required to avoid asserting $four_cells_valid$ signal when a new cell valid comes. For example, in Fig. 4.9, cell 80 is the first cell of a new line of cells and the state machine must not asserting the block valid signal at this point. From cell 81 onward, every valid cell comes would lead to a block valid.

To support this operation, the cell vectors are stored in a line buffer with 80+1 elements. Line buffer helps to save the memory resource because it only stores 81 out of 4800 cell vectors to calculate the normalization for 4661 blocks. Another advantage is that the pipeline can access cell vector immediately without requiring any external memory access. Finally, the pixels are calculated on the fly to avoid the latency cost of buffering them in the external memory as it is usually done in SW-only implementations.

Having asserted the signal $four_cells_valid$, the SQ stage calculates the square of all thirty-six bins separately (Fig. 4.11). Each input bin is a 15-bit width integer and $cell_binxx[14:0]$ signal represents bin x of cell x. As illustrated at the top of the Fig. 4.11, these bins are buffered in the pipeline using flip-flops until the MULTIPLY stage. The maximum square value for each bin can be stored in 30 bits.

At the ADD stage, all square values are added bin-wise as in Equation 4.8. It means that *sum_binx* represents the sum for bin0, bin1, bin2, bin3, bin4, bin5, bin6, bin7, and bin8. Since there is constant 1 in the Equation, the number of bits to represent the sum is 33.

$$sum_binx = \sum_{c=0}^{c=3} sq_cx + 1 \tag{4.8}$$

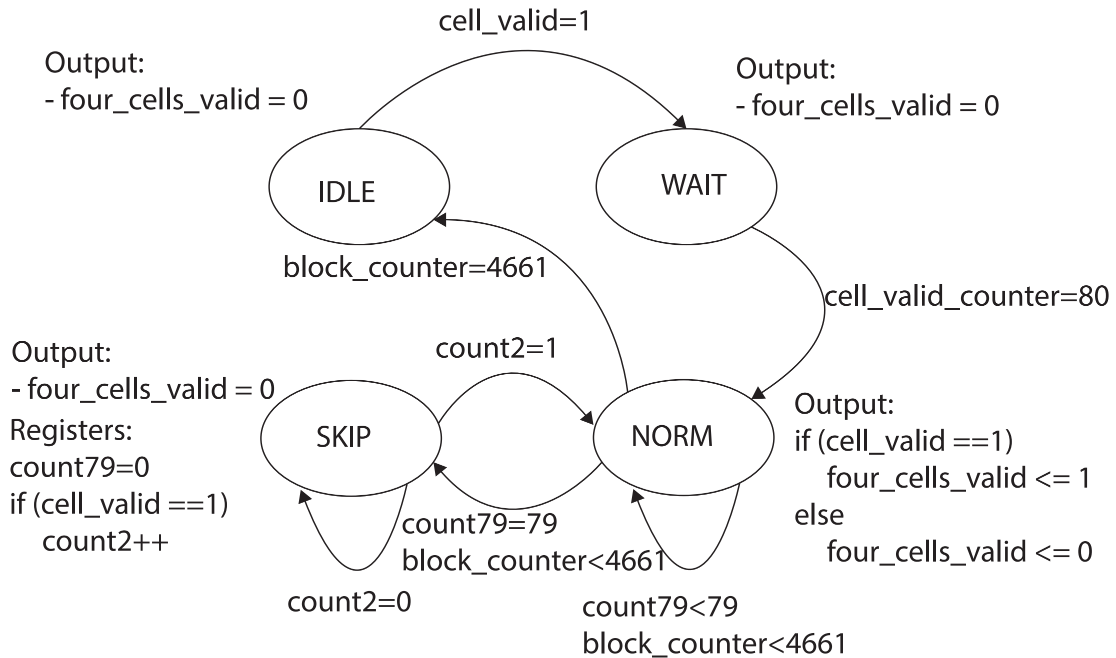


Fig. 4.10 Normalization finite state machine

Next, SQRT stage calculates the square root of these sums and generates the denominator of Eq. 4.7. It is necessary to instantiate nine square root modules to calculate the nine bins in parallel. The Integer Square Root IP from the library of IntelFPGA is used for this purpose. The output result is a 16-bit width integer.

Instead of dividing, Eq. 4.7 can alternatively be done by inverting and multiplying. The INVERSE stage efficiently does two tasks: converting integer to fixed-point and inverting. This is done by Eq. 4.9, where the numerator is shifted left by 16 bits. Therefore, the result of this stage is not the inversion, but the inversion which is shifted left by 16 bits. It equals to the inversion multiply by 2^{16}. The result, which is 17-bit width, can also be interpreted as a fixed-point number with 16 fractional bits.

$$inv_binx_fixed = \frac{1 << 16}{sqrt_binx} \tag{4.9}$$

The final stage carries out thirty six multiplications in parallel between 15-bit width hog bin and its corresponding *inv_bin_fixed*. This stage implements Equation 4.10, where *norm_xx_fixed* represents normalized feature of cell x and bin x. The output of this stage

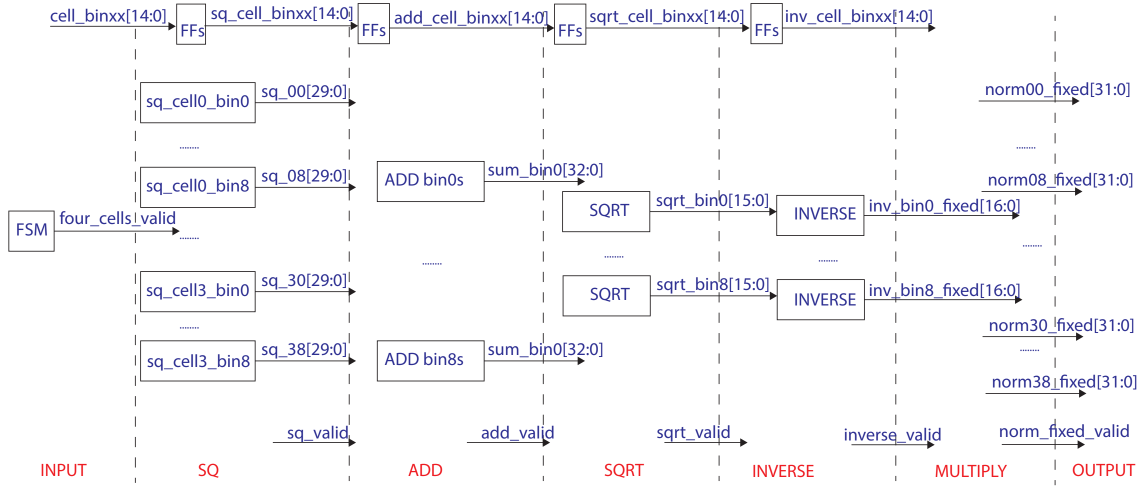


Fig. 4.11 Normalization pipeline

are 32-bit width fixed-point numbers with 16 fractional bits. The final normalized hog vector of a block is a vector of 36 fixed-point numbers.

$$norm_xx_fixed = inv_cell_binxx \times inv_binx_fixed \tag{4.10}$$

4.3 SVM Classifier pipeline design

The classifier pipeline essentially implements Equation 3.21. Therefore, the input to the pipeline are the SVM weights $\vec{W}$, the bias value b, and the HOG vector of a window $\vec{x}$. The size of a window is chosen to be $96x128$ pixels which equals to $7x15$ blocks. This size is sufficient to detect pedestrians with 80-pixel height as analyzed in 3.2. The windows are illustrated in Fig. 4.13. The output of the pipeline is the score of each window. This score is then compared against a given threshold to determine the final prediction if a pedestrian is present or not. This process is repeated for every window in the input image.

The key factor making an SVM classifier a long latency path in software is the sliding window task. This concept is illustrated in Fig. 4.12. The window buffer contains $7x15$ blocks of hog vectors that is used to convolve with the SVM weights stored in ROM memory. These 105 blocks consist of 3780 fixed-point numbers The convolution output is added with the bias b, provided by the trained model, to generate prediction score or confidence value.

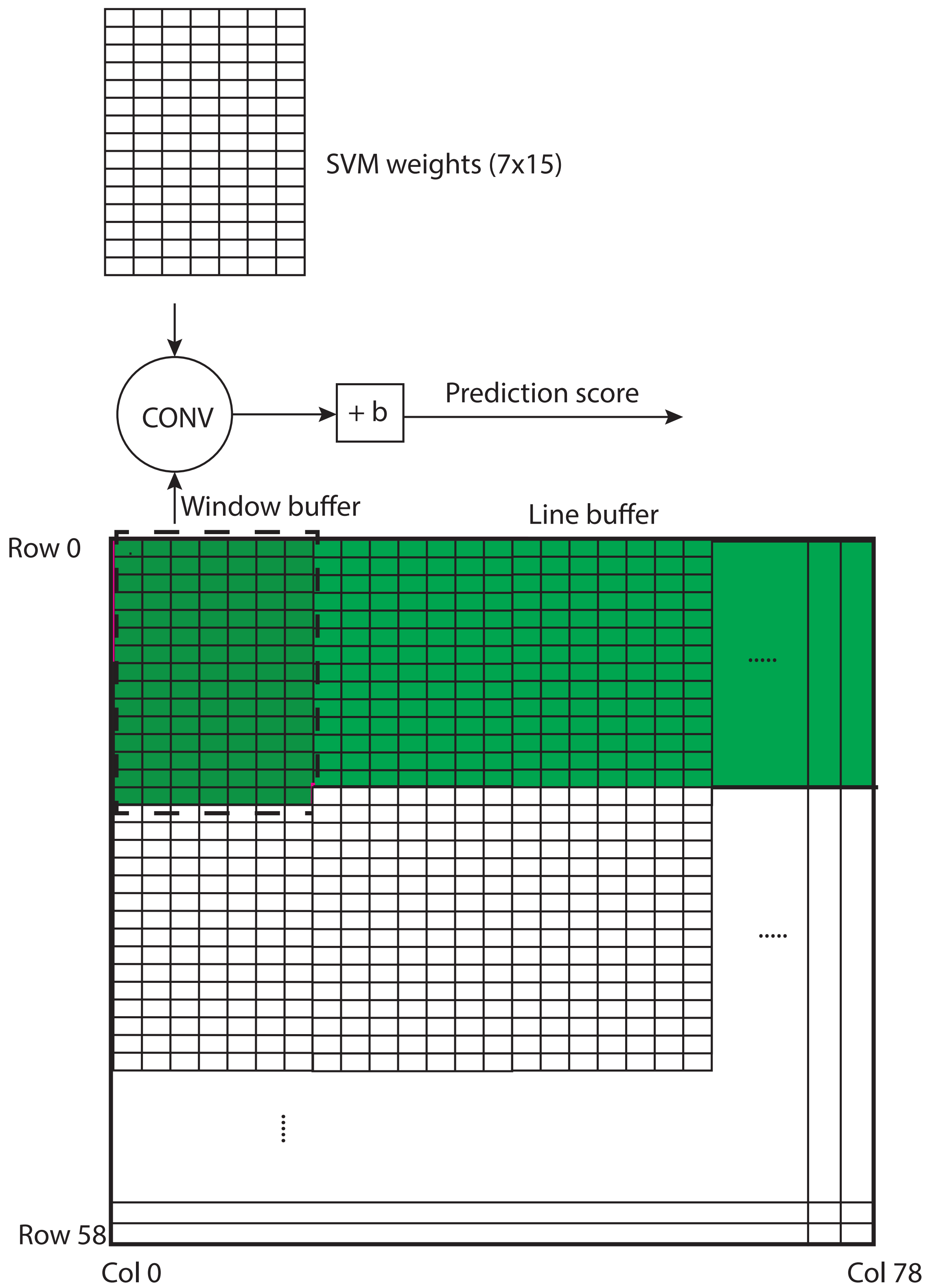


Fig. 4.12 Sliding-window and convolution illustration.

While the SVM weights are unchanged once the model is trained, content in the window buffer changes every time the line buffer shifts in a new block. The window buffer will be right shifted one column at a time until the end of the row. In this configuration, since the window buffer's width is 7, there are a total of 73 positions to be shifted in a row. After that, the line buffer in the Figure will be shifted down one row and the same shifting process of the window buffer will be executed. Within this specific configuration, there are a total of 45 rows that the line buffer needs to scan through.

The detail accelerator of the classifier is presented in Fig. 4.14. From the hardware point of view, there are two main problems to tackle to speed up the execution time compared to software implementation.

First, since blocks are generated sequentially, it is more efficient to process every block immediately after it is ready instead of waiting for the whole set of 105 blocks. Therefore, i will use the line buffer as described in Fig. 4.12. Its advantage is two-fold. On one hand, it saves memory resources because it only needs to store fourteen rows (+ 7 vectors) instead of all the blocks of an image. On the other hand, its access latency will be only one clock cycle.

Second, since a block is part of several window buffers, every block will be multiplied with different weights depending on the position of the window under calculation. Figure 4.13 shows that the black block belongs to several windows and it has different relative positions in those windows. Therefore, in this design, each block will be multiplied with all the corresponding weights and the resulting products will be accumulated to the appropriate windows.

The microarchitecture of the classifier is described in Fig. 4.14. The pipeline registers are not shown for the sake of clarity. Each hardware component will be described in the following paragraphs.

- MAIN CONTROLLER: FSM that controls the whole classifier. First, the FSM will check if a new block is available before fetching it to the pipeline. Knowing the block position (because blocks come in raster scan order), the FSM generates appropriate addresses to access ROM and RAM memory for the multiplication to the weights. The multiplication result is stored at different RAM positions depending on which window the block belongs to.

- ROM: Stores all the weights. The size of this ROM is $3780x10$ bits. Each element of the weight vector is represented by a 10-bit fixed-point, with 8 fractional bits. To generate the weight vector, several models have been trained and tested with different configurations using Caltech dataset. If the pre-trained model is changed, ROM must be reloaded with the new weight vector set.

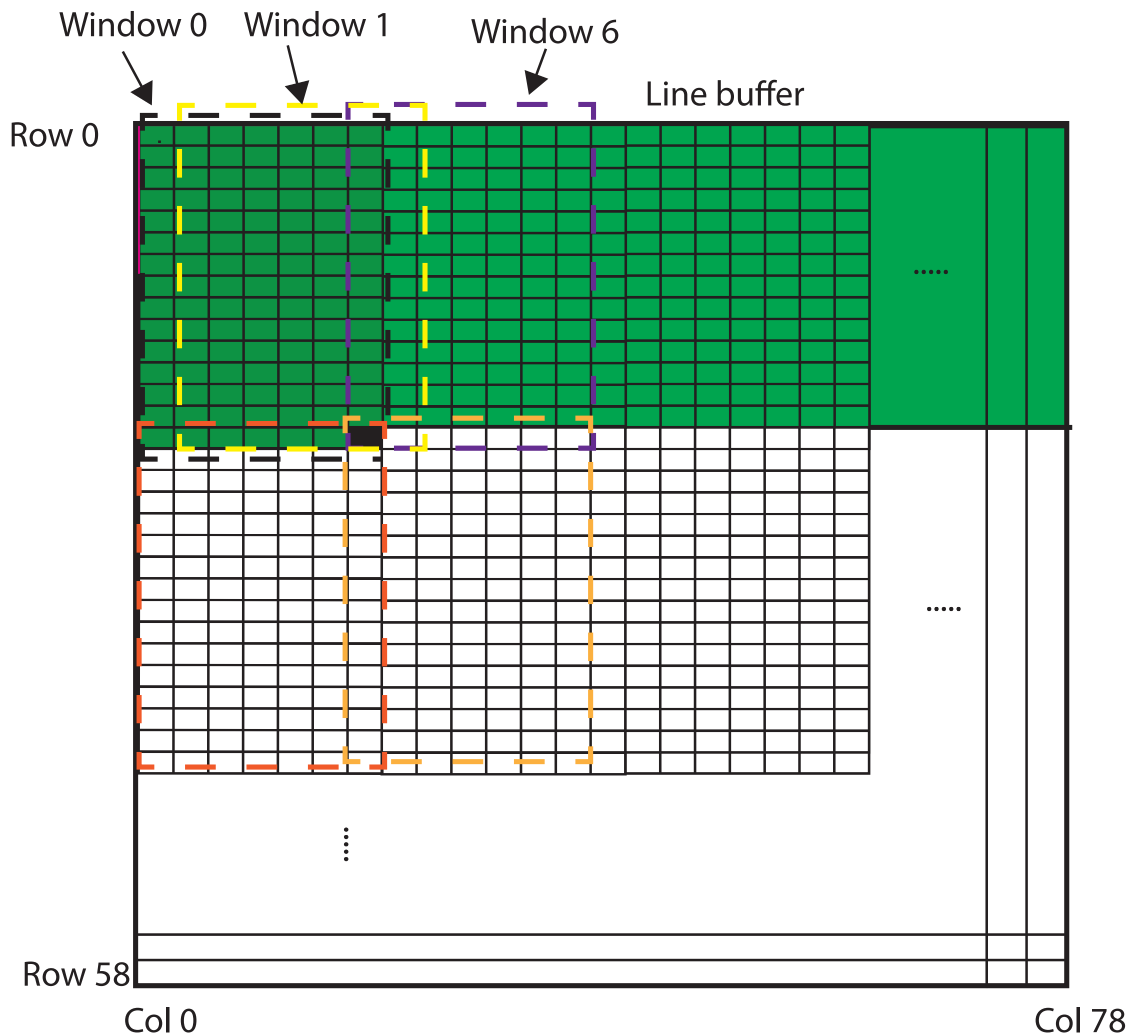


Fig. 4.13 Illustration of a block being part of different windows

- RAM: there are two RAM instances (shown in Fig. 4.14) to distinguish between reading and writing. Physically, only one dual-port RAM module is instantiated in the design since read and write access to same address will never happen. The memory, with a size of $30x73x19$ bits, stores temporary sums for final confidence values. Each resulting confidence value of a detection window is a sum of 105 partial sums. Each word stored in RAM is 19-bit width, 12-bit for the partial sum and a 7-bit for the counter. The counter is used to signify that the detection window's confidence value is valid. The *win_done* signal is active when 105 partial sums of a detection window are fully accumulated. To optimize on-chip memory usage, the memory location storing that window's value will be reused for other detection windows. Then, the design

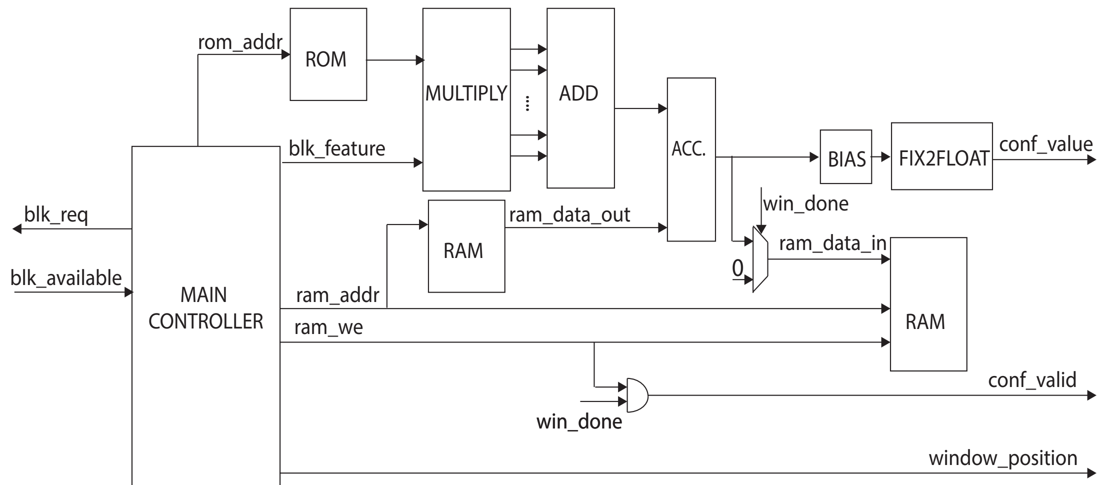


Fig. 4.14 SVM classifier hardware block diagram

uses only $30x73$ RAM locations to store the temporary sums of the $45x73$ detection windows.

- MULTIPLY: This module takes a HOG block and multiplies it with appropriate elements of the weight vectors stored in the ROM memory. One-cycle multiplication will generate 36 products because a block contains 36 elements. Depending on the position of the block, it might belong to multiple detection windows. The complete processing of any specific block (if that block belongs to 105 detection windows) takes 105 clock cycles.

- ADD: This module simply sums up 36 products from the MULTIPLY module.

- ACC.: Since a detection window's confidence value is the sum of 105 partial values, this module adds the accumulated value stored in the RAM memory with new partial sums coming from the ADD module.

- BIAS: This module adds the bias value (from the SVM trained model) to generate the final confidence value in fixed-point representation.

- FIX2FLOAT: Fixed-point confidence values are converted to 32-bit floating-point numbers by this block.

  From the right side of Fig. 4.14, we can see that each confidence value is accompanied by a valid signal and an address indicating the position of that detection window in the image. The window position is used by the HPS software to draw the rectangular

if the confidence value is higher than the threshold or, in other words, a pedestrian is detected.

4.4 Results

This sections presents the main results achieved concerning functional accuracy and computational performance that are later compared to the state-of-the-art implementations.

4.4.1 Accuracy

In the literature, there are two methods for counting the number of false positives. One is false positives per image (FPPI), based on the entire image. Since the detection window slides through the whole image to convolve and obtain detection results, this method creates more false positives. To reduce those, it requires a Non-Maximal Suppression (NMS) algorithm (section 3.2. The NMS algorithm picks the highest IOU detection result and considers the remaining results, which relate to the same pedestrian, as false positives.

Another way of counting false positives relates to non-sliding detectors. In this case, a false positive only occurs when the detector finds a pedestrian in a window that has no pedestrian. This is called false positive per window (FFPW). One can infer that the FFPW potentially has less false positives than FFPI while FFPI is more suitable for real-world scenarios.

In this thesis, the accuracy of the implementation is measured with FFPI using a golden model written in C/C++. It can be found at the link [139]. The software model achieves almost the same performance as the original HOG's performance testing with Caltech dataset as shown in Figure 4.15. The lower the curve, the better the accuracy. In autonomous car applications, the number of false positive per image should be equal or less than 0.1. At the right side of the curve, where the number of false positive per image is equal or bigger than 1, even though our model provides better miss rate, it is not applicable for autonomous cars because the number of false positive per image is too high. The evaluation is carried out with the reasonable setting where the detector only looks for higher than 50 pixels and partially or non-occluded pedestrians [116].

The hardware implementation is then compared against the golden model for functional verification and accuracy testing. The checking points are $delta_x$ and $delta_y$, the gradients (magnitude and orientation), bin values, cell HOG vector, and block HOG vector. Based on the testing results, the similarity between the hardware's block HOG vector output and that of the software model is up to two digits after the decimal point. It is worth noting that the

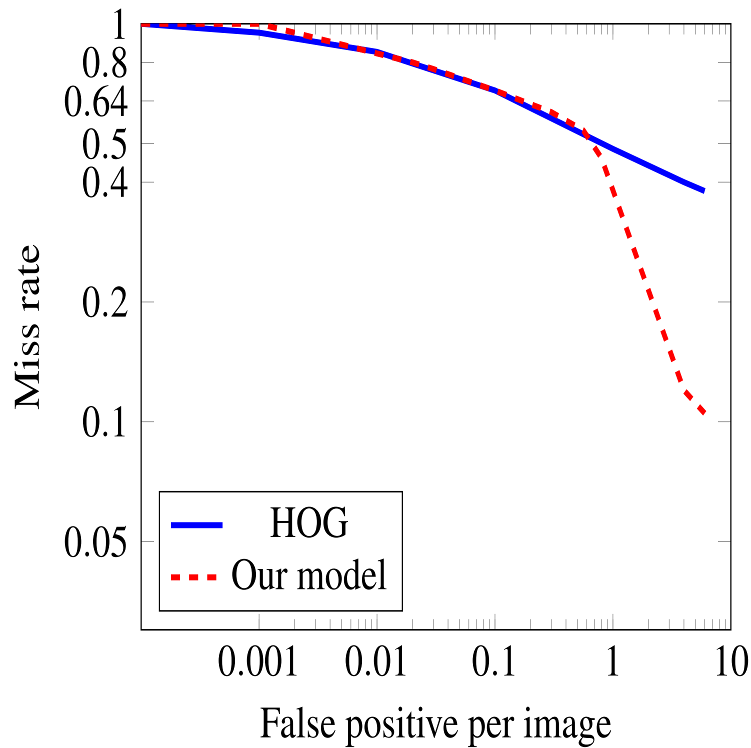


Fig. 4.15 Comparison of our model with the standard HOG on Caltech pedestrian dataset

software model uses floating-point numbers while the hardware design works with integers and fixed-point numbers.

4.4.2 Latency

Latency of the system is defined as the sum of two components: (1) the delay time to get a full frame and (2) the latency to process the ending pixel to have the prediction score for the last detection window in the image. The first component is exactly the Frame time parameter which can be found in appendix A. The Frame time for the configuration in this design is 89 ms.

The second part of the latency is broken down in Fig. 4.16. The delay of each stage in the pipeline is shown below every functional block. This delay is specific for the case when the pipeline processes the ending pixel of the input frame. The total latency is 122 cycles, which is 2,440ns, for the final pixel of any frame. This part of latency is very small (compared to the first part). The reason for this is that the latency of the HOG+SVM pipeline is hidden in the latency of receiving pixels from the sensor. This is inherently a good result since it depends on the sensor acquisition speed rather than in the computation.

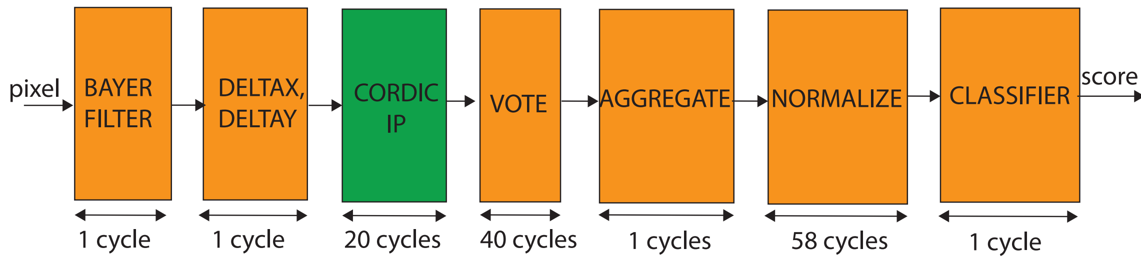


Fig. 4.16 Latency of the whole pipeline

Therefore, the latency of the HOG+SVM system is 89 ms at the working frequency of 50 MHz. It is less than 1/10 second. There should be enough time margin to give appropriate control on the car if pedestrian is detected.

4.4.3 Throughput

The throughput of the system concerning frame throughput is the inverse of the latency. Therefore, at 50 MHz, the system achieves a throughput of 11 FPS. This throughput is not as high as the real-time throughput target of the thesis. However, both the camera's pixel clock and the pipeline working clock can be upscale. For instance, if the pixel clock is 140 MHz, the throughput of the system would achieve approximately 31 FPS. The HOG+SVM pipeline can also work at 140 MHz since its maximum frequency is up to 162 MHz.

4.4.4 Comparison with existing implementations

Table 4.2 compares our implementation to the existing implementations. Regarding FPGA resources, the two most critical resource on FPGAs are on-chip memory and LUTs. Even though the design in this thesis does not use external memory, it consumes the least on-chip memory resource except for the one in [140], which reports zero memory usage. Since input resolution is different among the implementations, the memory bit per pixel information is extracted for fairly comparison. Accordingly, our design achieves one on-chip memory bit per pixel, which is among the top four best designs. The reason for this is that every input pixel or intermediate result is processed on the fly using line buffers and there is not any buffer for input frames or intermediate result. Moreover, line buffers help the design to be scalable with higher resolution input because the size of the line buffers only depend on the width of input images.

Regarding the number of LUTs, the implementation in [141] is the most efficient, followed by the one in [142] and then the design presented in this thesis. The reason is that both implementations ([141], [142]) target low resource utilization by simplifying some computa-

tional operations. In their voting part, magnitudes are voted to only one unique bin without interpolation. To ensure the accuracy, the design in this thesis used linear interpolation to split a magnitude into two bins as presented in section 3.4.3. Furthermore, in [141], all the calculations use integer numbers.

About DSPs usage, this implementation uses 38 DSPs which is the third in the ranking. The best one only uses four DSP blocks [140].

In terms of processing speed, this design takes 89 ms to detect a frame, which corresponds to a throughput of 11 FPS. It is worth noting that this latency is also the time to receive a frame from the image sensor. The pixel clock from the image sensor in this configuration is 50 MHz and it can be boosted to run at higher frequency such as 100 MHz or 150 MHz. The HOG+SVM engine in this design can work at a maximum frequency of 162 MHz.

Since working frequency is different among implementations, the pixel per clock period information is extracted to compare the efficiency of the hardware pipelines. This design achieves the second to the best efficiency with 0,068 pixels per clock.

With respect to power consumption, this design consumes the lowest power in the Table. The key reason for this is that the design only uses on-chip memory and avoids external memory access. A second reason is that the design uses less resource compared to other designs, which leads to lower power consumption. There are also differences concerning the measurement method. The authors in [143] used Xilinx Xpower Analyzer software to estimate the power consumption (which can avoid technologically dependent contributions) while we obtain the result from an energy meter model FHT-999. Anyway, the design works at 50 MHz and its throughput is only 11 FPS.

In terms of energy efficiency, measured by the number of FPS per watt, our implementation is the third-best after [95], [143].

Table 4.2 Comparison with the state of the art

Implementation	[144]	[141]	[95]	[140]	[143]	[145]	[146]	this work
Year	2013	2013	2015	2015	2015	2018	2020	2019
Hardware	Virtex 6	Virtex 5	Virtex 6	XC7Z020	Virtex 7	Cyclo. IV	Kintex UltraScale	Cyclone V
Node	40 nm	65 nm	40 nm	28 nm	28 nm	60 nm	20 nm	28 nm
Freq.(MHz)	NA	266	150	82,2	266	150	150	50
Frame size	1024x768	1920x1080	640x480	1920x1080	1920x1080	800x600	800x600	640x480
Latency	4,88 ms	<150 μs	44 ms	25,2 ms	NA	NA	9 ms	89 ms
Power (W)	182	NA	37	NA	19	NA	NA	9
Energy (J/frame)	14	NA	0,54	NA	0,45	NA	NA	0,8
FPS	13	64	68,2	40	42,7	162	115	11
Memory (Kb)	3.744	1.188	13.738	0	4.079	344	756	317
LUTs	108.518	5.188	184.953	21.297	30.360	16.060	7.804	13.464
DSPs	138	49	190	4	364	69	36	38
FFs	120.576	5.176	208.666	NA	48.576	7.220	NA	17.117
Pixels/clock	NA	0,0005	0,0003	0,0009	0,0003	0,0009	0,37	0,068
FPS per watt	0,07	NA	1,84	NA	2,25	NA	NA	1,22
Memory/pixel (bit)	5	0,6	46	0	2	0,7	1,6	1

4.5 Summary

In conclusion, a low power pedestrian detection has been implemented and tested on Cyclon V FPGA. The design achieves the lowest power consumption in the SOA. Its throughput is only 11 FPS due to the long latency of getting a frame from the image sensor. Boosting the pixel clock can increase the throughput of the system. The design is optimized to use low resource to save the power and resource for future integration with other algorithms on autonomous cars.

However, the optimization in the design process makes it difficult to adapt to different configurations. Therefore, in the next chapter, a more flexible design is introduced for the same system using OpenCL.

Chapter 5

HOG/SVM pedestrian detection implementation using OpenCL

5.1 OpenCL programming model

OpenCL is a framework for parallel programming on heterogeneous systems like CPUs, GPUs, DSPs, FPGAs, and other hardware accelerators. It was initiated by Apple and later standardized and released in 2008 by Khronos Group. For FPGAs, it supports generating HDL code from OpenCL code and creating communication channels between FPGA accelerators and the host processor. This section introduces the design flow of this framework for FPGAs.

The OpenCL programming model in this thesis is illustrated in Fig. 5.1. This model is specific to the configuration that is used in this study. The host is an Intel Core i7 CPU. It has 8 cores, running at 3.5 GHz. The FPGA board is DE5Net with Stratix V chip. The board communicates with the CPU through PCIe interface.

In this programming model, the host code and the accelerator code are developed in the same environment. These source codes are independent of hardware platforms. The host code will be compiled and executed in the host, in which there are invocations to the run-time library functions to interact with the accelerators and the memory on the FPGA side. The run-time library is developed by FPGA vendors.

The acceleration code is written in accel.cpp (Fig. 5.1). The OpenCL framework uses High Level Synthesis (HLS) to generate hardware from C/C++ source code. It is compiled into hardware kernels on FPGA fabric which will be invoked by the host code. The host interacts with the hardware kernels and the FPGA's memory through the PCIe interface. First, the accelerator code is compiled into Verilog code. Then, the FPGA bitstream will be generated by synthesis and place & route tools.

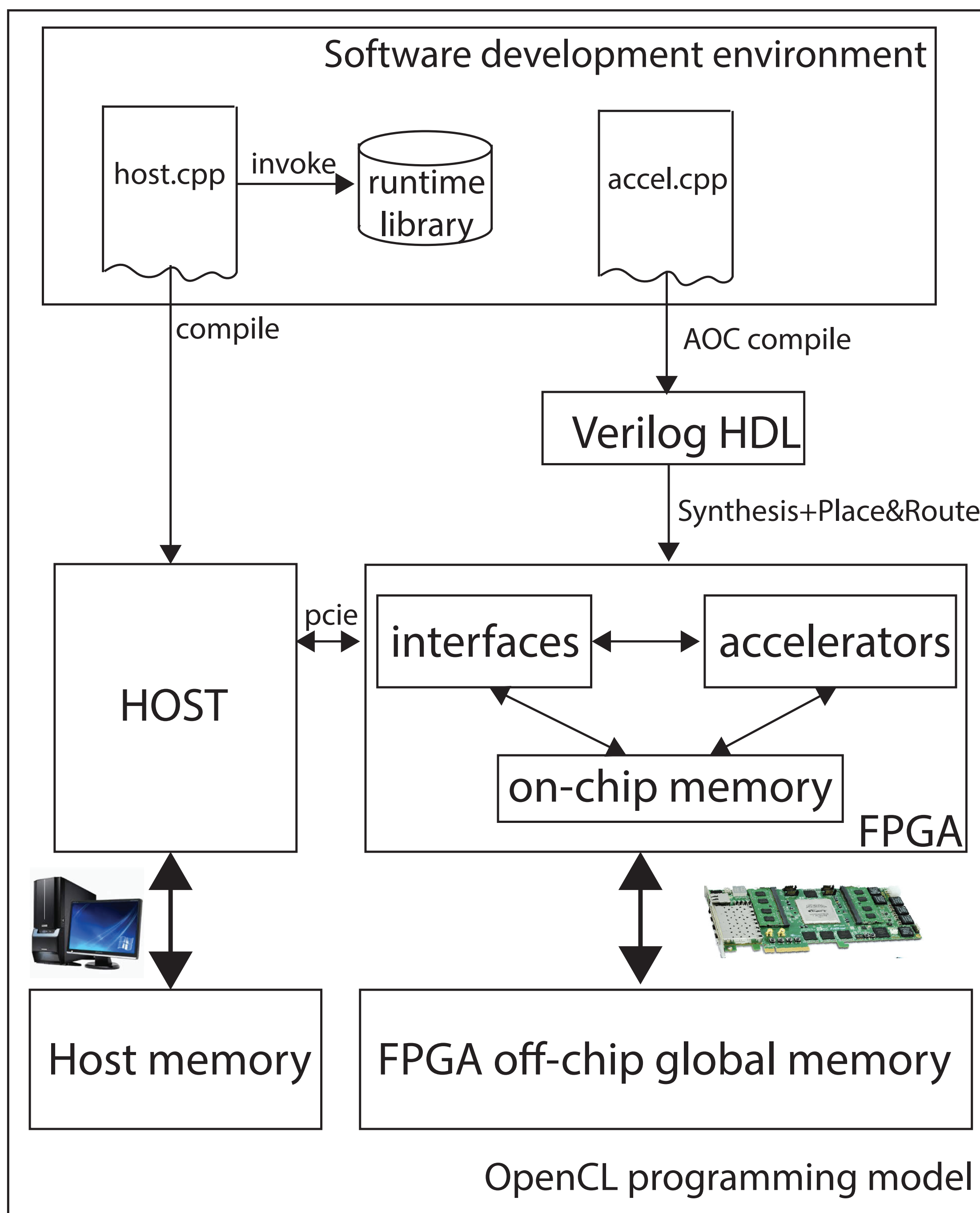


Fig. 5.1 OpenCL programming model

A standard OpenCL memory model is described in Fig. 5.2. A host might connect to one or more compute devices, FPGAs in this context. The connection interface is usually PCIe. Within a compute device, there might be one or more work groups executing in parallel.

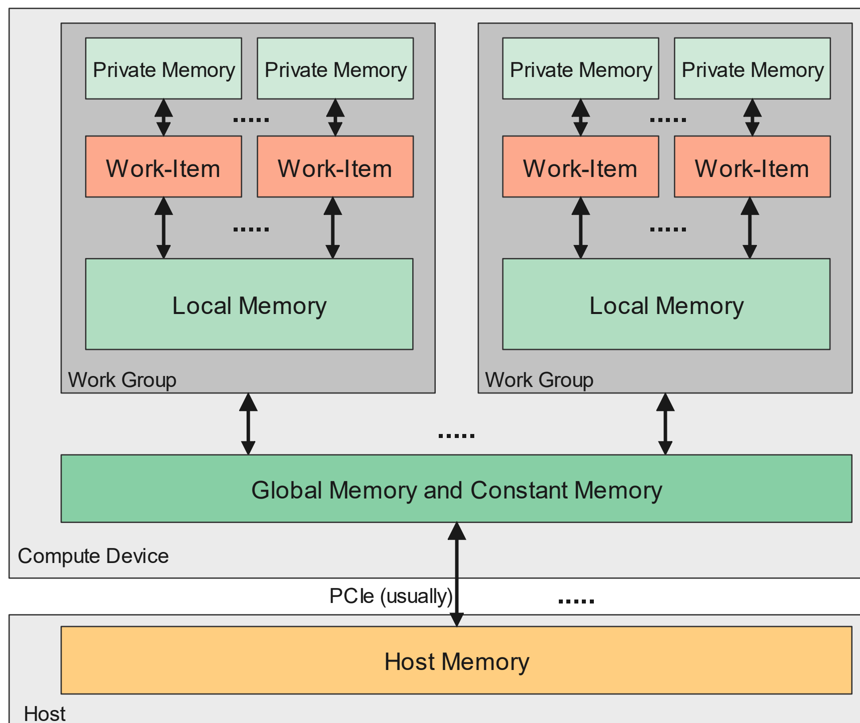


Fig. 5.2 OpenCL memory model

A work group is formed from one ore more work items. The work items are the hardware kernels and they usually implement same function but on different data. A kernel has the capability to access to four disjoint address spaces. They are described as follows:

- Private memory: It is private to a work item. It is fast (one clock cycle) and small size. It is usually implemented by registers or on-chip memory. Size of this memory is usually thousands of words (32 bits) per work item. A variable can be declared with private qualifier to be allocated in this memory region. Function arguments or local variables of functions shall also be allocated in private address space. Too much private variables declared in a kernel would make some of them to be located in other address spaces such as local or global memory even thought they are still private.

- Local memory: This memory region is private to a work group. It is shared across all work items in the work group. A variable can be explicitly defined to be in local memory by local qualifier. This memory space is implemented by registers or on-chip memory of FPGAs. This memory region is not visible from the host. Hardware kernels are responsible to transfer data between local and global/constant memory. Accessing

this space is still very fast but it is slower than private space because it is shared for multiple work items.

- Constant memory: This is the address space that stores variables defined at global scope and accessed as read-only variables inside kernels. This address space is accessible to all kernels.

- The global memory, residing in the FPGA side, can be accessed by both the host and the accelerators. It is implemented by both off-chip memory, usually SDRAM chips, and on chip memory. The global memory region is shared across all work items in all work groups. Global variables can be declared using global qualifier.

On the host side, the host memory can only be accessed by the host. Basically, the host first allocates the necessary data from host memory into global/constant memory using the API functions. Then, OpenCL kernels, invoked by the host code, access the FPGA's memory for input and output data. Finally, output data are transferred to the host memory.

5.1.1 Optimization techniques

The Intel OpenCL offline compiler recommends users to structure OpenCL kernels as single work items. The compiler optimizes a single work item kernel by pipelining its loops. The document IntelFPGA SDK for OpenCL Best Practice Guide [147] presents good design practice for a single work-item kernel. All the kernels in this thesis follow this guidance. In this section, the optimization techniques used in the kernels for memory and computation operations are described.

Memory operations

Accessing on-chip memory is much faster than external global memory. One way to achieve on-chip memory latency is to make variables private.

One way to achieve this is to put constant keyword for appropriate variables so that an on-chip cached memory is created. For example, in Listing 5.1, the constant *my_array* with preinitialized value will be stored in on-chip ROM.

```
1 __constant int my_array[8] = {0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7};
2 __kernel void my_kernel (__global int * my_buffer)
3 {
4   size_t gid = get_global_id(0);
5   my_buffer[gid] += my_array[gid % 8];
```

```
}
```

Listing 5.1 Example of constant keyword [1]

Another technique that avoids memory dependencies is to define restrict keyword in memory pointers that point to separated memory regions. For example, in Listing 5.2, A and B are two pointers pointing to non-overlapping regions in global memory.

```
__kernel void myKernel (__global int * restrict A,
                        __global int * restrict B)
```

Listing 5.2 Example of restrict keyword [1]

Those are compiler-based optimizations. Architecturally, the design uses on-chip memory to implement line buffers mentioned in Chapter 4. With line buffers, kernels are optimized to minimize the number of external memory accesses. An implementation of a line buffer is shown as an example in Listing 5.3. This code example implements a line buffer of two consecutive rows plus three elements of the next row. The length of a row is the number of pixels in a row of an input image. This length is parameterized as *COLS*

```
__kernel
void camera_gradient(global unsigned short * restrict frame_in,
           const int iterations, global float * restrict mag_out, global
    float * restrict orien_out)
{
    // Pixel buffer of 2 rows and 3 extra pixels
    unsigned short rows[2 * COLS + 3];

    // The initial iterations are used to initialize the pixel buffer.
    int count = -(2 * COLS + 3);

  while (count != iterations) {
        // Each cycle, shift a new pixel into the buffer.
        for (int i = COLS * 2 + 2; i > 0; --i) {
            rows[i] = rows[i - 1];
        }
        //Other lines of code
        rows[0] = count >= 0 ? frame_in[count] : 0;
        count++;
    }
}
```

Listing 5.3 Implementation of a line buffer

Computation operations

The key parts in OpenCL code where HLS finds opportunities to increase design's performance are loops. There are two approaches to optimize loop execution latency. One method is to fully unroll them so that all iterations are executed in parallel when possible. This reduces the loop latency and improves the performance yet consumes more resources. Xilinx's compiler automatically unroll loops that have number of iterations equal to or less than 64 [148].

The second method, which not only accelerates the loop execution but also consumes resource effectively, is to pipeline the loop. The latency of a loop, formulated by Zohouri et al. [149], is represented in Equation 5.1, where P is the number of pipeline stages, L is the number of iterations, II is the initiation interval, and f is the clock frequency.

$$t = \frac{P + II(L-1)}{f} \tag{5.1}$$

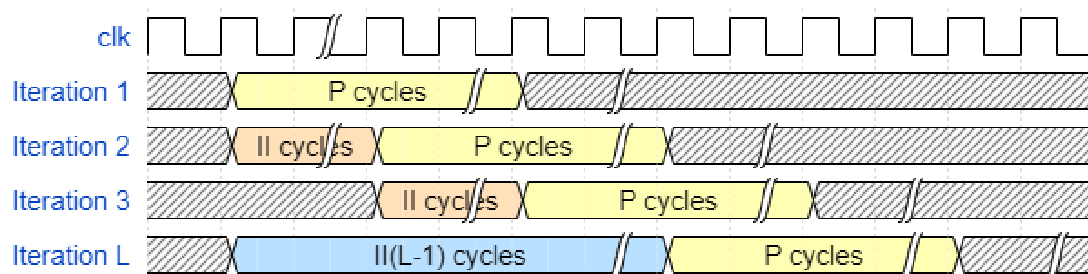


Fig. 5.3 Pipelined loop latency: L iterations

Initiation interval (II) is the number of clock cycles that the pipeline needs to wait before it can proceed to the next iteration. An optimal loop in OpenCL should have an II of 1. Figure 5.3 presents the execution order of a loop with L iterations, in which a new iteration is only executed II cycles after the previous iteration gets started.

Floating-point operations are usually found in loops; they usually traverse through multiple clock cycles and create data dependency. Data dependency makes the II greater than 1. To avoid this data dependency and achieve the lowest II, kernels are implemented in such a way that shift registers are inferred. The technique is described in [147].

For example, an unoptimized kernel is shown in Listing 5.4. The kernel accumulates all double-precision FP elements in the array *arr*. The size of the array is an input parameter. According to [147], every loop iteration takes 11 cycles to execute the adding operation. The current iteration requires the *temp_sum* value of the previous iteration. Therefore, there is a data dependency on the variable *temp_sum*.

```
__kernel void double_add_1 (__global double *arr,
                            int N,
                            __global double *result)
{
  double temp_sum = 0;
  for (int i = 0; i < N; ++i)
  {
    temp_sum += arr[i];
  }

  *result = temp_sum;
}
```

Listing 5.4 Floating-point adder without shift registers

To relax that data dependency between loop iterations, a shift register is inferred using the source code in Listing 5.5. The shift register *shift_reg* removes the data dependency on *temp_sum*. The size of the shift register should be bigger than the latency of the adder. Since the latency of the adder is 11 clock cycles, the length of the shift register is set to 12.

```
  //Shift register size must be statically determinable
  #define II_CYCLES 12

  __kernel void double_add_2 (__global double *arr,
                              int N,
                              __global double *result)
  {
      //Create shift register with II_CYCLE+1 elements
      double shift_reg[II_CYCLES+1];

     //Initialize all elements of the register to 0
     for (int i = 0; i < II_CYCLES + 1; i++)
     {
         shift_reg[i] = 0;
     }

     //Iterate through every element of input array
     for(int i = 0; i < N; ++i)
     {
         //Load ith element into end of shift register
         //if N > II_CYCLE, add to shift_reg[0] to preserve values
         shift_reg[II_CYCLES] = shift_reg[0] + arr[i];

         #pragma unroll
         //Shift every element of shift register
```

```
26          for(int j = 0; j < II_CYCLES; ++j)
27          {
28              shift_reg[j] = shift_reg[j + 1];
29          }
30      }
31
32      //Sum every element of shift register
33      double temp_sum = 0;
34
35      #pragma unroll
36      for(int i = 0; i < II_CYCLES; ++i)
37      {
38          temp_sum += shift_reg[i];
39      }
40
41      *result = temp_sum;
42  }
```

Listing 5.5 Floating-point adder with shift registers

5.2 OpenCL implementation

The OpenCL-based implementation for HOG+SVM is described in Figure 5.4. The key parts of the system are the host and the coprocessor which consists of the global memory and the FPGA configurable logic. The accelerated modules are implemented on FPGA logic and they are invoked by the main software running on the host. The global memory is the memory on the FPGA board. It stores images, gradients, HOG features, detection scores, and other intermediate data. The kernels read and write data from and to this memory as illustrated by the thick arrows between the global memory and the FPGA device.

The host software also accesses the global memory to prepare data for hardware accelerators. The access from software is shown as thick arrows between the host and the global memory.

First, the software writes input images into the global memory. Second, the histogram function in the host reads HOG features from the global memory which are generated by the histogram kernel. After that, the function re-arranges them in blocks and writes them back to the global memory. At the end of the pipeline, the classification function reads detection scores from the global memory to visualize detection results. The visualization task draws rectangulares at positions where pedestrians are predicted with corresponding confidence values.

In this thesis, the four hardware kernels are shown in the FPGA device portion of Figure 5.4. These kernels are implemented and tested on five different FPGA platforms, which are OpenVino Starter Kit, DE5Net, DevCloud PACS10, DevCloud PAC10, and HARP+Arria10. OpenVino Starter Kit is the board with the smallest resource; HARP Xeon+Arria has the biggest resource. The pattern we saw is that the bigger board spends more resource and achieves better latency. This is correct for all the kernels except for the histogram kernel. The implementation of the kernels are presented in the following sections.

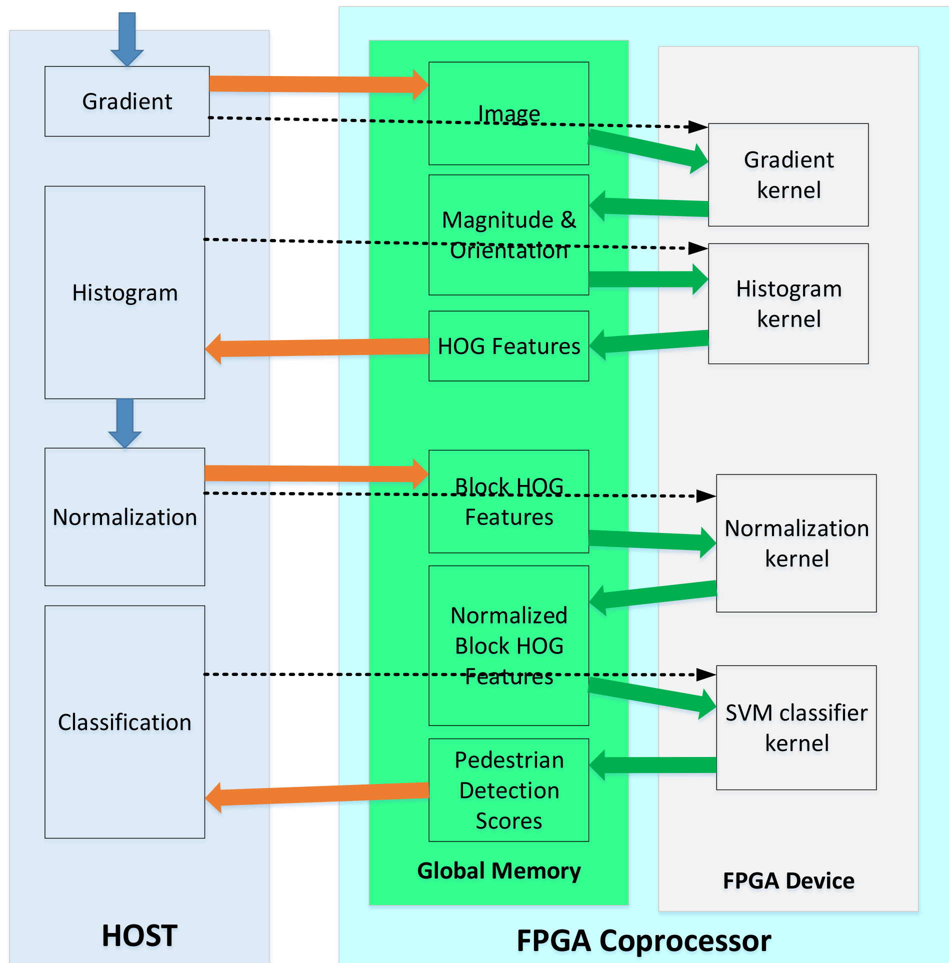


Fig. 5.4 Hardware-software inter-operation based on OpenCL

Gradient kernel

Gradient kernel loads pixels from global memory, calculates the magnitude and orientation gradients, and stores back into global memory. The number of pixels for each frame

is 640x480. The 1D-centred mask, shown in Table 3.5, is chosen as the convolution mask because it is both simple and effective.

The OpenCL code for Gradient kernel is shown in Listing 5.6. The number of iteration is set by the host code through the constant parameter *iterations* and it is equal to the input frame's size 640x480. The kernel writes back magnitude and orientation gradients of every pixel into global memory. Since these two memory regions and the input frame region are totally separated, the pointers to them are defined with *restrict* keyword. Shift registers are inferred in this kernels for variable *rows*. Floating-point numbers are used to represent gradients to ensure the equivalence with the software model. The kernel's initiation interval is reported to be 1, which is an optimal number.

```
#include "../host/inc/defines.h"
// Gradient filter kernel
// frame_in and frame_out are different buffers. Specify restrict on
// them so that the compiler knows they do not alias each other.
__kernel
void camera_gradient(global unsigned short * restrict frame_in,
            const int iterations, global float * restrict mag_out, global
     float * restrict orien_out)
{
  double pi =     3.14159265359;
    // Pixel buffer of 2 rows and 3 extra pixels
    unsigned short rows[2 * COLS + 3];

    // The initial iterations are used to initialize the pixel buffer.
    int count = -(2 * COLS + 3);

  while (count != iterations) {
        // Each cycle, shift a new pixel into the buffer.
        // Unrolling this loop allows the compile to infer a shift
   register.
        #pragma unroll
        for (int i = COLS * 2 + 2; i > 0; --i) {
            rows[i] = rows[i - 1];
        }
        rows[0] = count >= 0 ? frame_in[count] : 0;
        int x_dir = 0;
        int y_dir = 0;
    unsigned int x_p1, x_m1, y_p1, y_m1;
    unsigned int current_row = (count-COLS-1)/COLS;
    unsigned int current_col = (count-COLS-1)%COLS;

    x_p1 = (current_col<(COLS-1))? rows[COLS]:rows[COLS+1] ;
```

```
31     x_m1 = (current_col >0)? rows[COLS+2]: rows[COLS+1];
32     y_p1 = (current_row <(ROWS−1))? rows[1]: rows[COLS+1];
33     y_m1 = (current_row >0)? rows[2*COLS+1]: rows[COLS+1];
34     x_dir = (x_p1&0xff)−(x_m1&0xff);
35     y_dir = (y_p1&0xff)−(y_m1&0xff);
36 
37     float grad = (float) sqrt(x_dir*x_dir + y_dir*y_dir);
38     float orien = atan2 ((float) y_dir,(float)x_dir);//value from −
    pidiv2 to pidiv2
39     if (orien <0)
40       orien = orien + pi; //all orientations are from 0 to PI
41 
42     if (count >= COLS+1) {
43       mag_out[count−COLS−1] = grad;
44       orien_out[count−COLS−1] = orien;
45         }
46         count++;
47     }
48 }
```

Listing 5.6 OpenCL code for the Gradient kernel

The kernel is tested on different FPGA platforms for performance comparison. The result is shown in Table 5.1. The HARP Xeon+Arria 10 platform has the smallest execution and transfer time.

Histogram kernel

Having the magnitude and orientation in global memory, the histogram kernel reads these values and writes out HOG feature vector of every cell. Therefore, the number of iteration in this kernel is the number of cells. A HOG vector of a cell consists of 9 float values, corresponding to 9 bins. In the first place, a 7x8 line buffer is implemented to read in the gradients serially. The input gradients are float numbers, but they are converted into fixed-point numbers to save hardware resource. The initiation interval, however, is up to 31, which imposes very high latency.

Next, the line buffer is removed from the kernel to reduce the II. This kernel reads the gradients directly from the global memory. From Table 5.2, ALUT and FF resource are reduced almost a half; the RAM and DSP resource are the same. The initiation interval, however, is still very high.

The third version of this kernel uses float data type as it is more accurate than a fixed-point. The hardware resource overhead is 5%, 3%, and 2% of the total available ALUTs, FFs, and

RAMs, respectively. The initiation interval of this float-based version is reduced to 24. The reason for this improvement is the removal of the converters from float-point to fixed-point number and vice versa.

Based on the floating-point version, shift registers inference is added to further reduce the initiation interval to 1 and increase the throughput of the kernel. The ALUTs and FFs resource usage increases accordingly. The overhead, as shown in Table 5.2, is acceptable since the minimum initiation interval is achieved. The code of the kernel is presented in Listing 5.7. In this kernel, I write the source code so that nine different shift registers are inferred for the nine bins.

The best version of the histogram kernel in terms of II is compiled and compared with different FPGA platforms as in Table 5.3. With this kernel, the DE5Net board provides the lowest latency.

```
1  #define FLOAT_ADD_CYCLES 6
2  #include "../host/inc/defines.h"
3
4  __kernel
5  void hog_vector(global volatile float * restrict mag_in, global volatile
       float * restrict orien_in,
6              const int iterations, global float * restrict hog_vector_out)
     {
7     const float pi9_inv = 2.86478897565;
8    const float pi18    = 0.17453292520;
9    const float pi9    =  0.34906585040;
10   int count=0;
11   while (count != iterations) {
12         float bins[9]={0};
13     float grad_mag;
14     float grad_orien;
15     float shift_bin1[FLOAT_ADD_CYCLES+1];
16     float shift_bin2[FLOAT_ADD_CYCLES+1];
17     float shift_bin3[FLOAT_ADD_CYCLES+1];
18     float shift_bin4[FLOAT_ADD_CYCLES+1];
19     float shift_bin5[FLOAT_ADD_CYCLES+1];
20     float shift_bin6[FLOAT_ADD_CYCLES+1];
21     float shift_bin7[FLOAT_ADD_CYCLES+1];
22     float shift_bin8[FLOAT_ADD_CYCLES+1];
23     float shift_bin9[FLOAT_ADD_CYCLES+1];
24
25     //inititalize all elements of the shifft registers to 0
26     #pragma unroll
27     for (int i=0; i<FLOAT_ADD_CYCLES+1; i++)
```

```
28    {
29      shift_bin1[i]=0;
30      shift_bin2[i]=0;
31      shift_bin3[i]=0;
32      shift_bin4[i]=0;
33      shift_bin5[i]=0;
34      shift_bin6[i]=0;
35      shift_bin7[i]=0;
36      shift_bin8[i]=0;
37      shift_bin9[i]=0;
38    }
39
40        int cell_row= count/NUM_CELLS_X;
41    int cell_col= count%NUM_CELLS_X;
42        int frame_row = cell_row*CELL_H;
43    int frame_col = cell_col*CELL_W;
44        //#pragma unroll
45        for (int i = frame_row; i < frame_row+8; ++i) {
46            //#pragma unroll
47            for (int j = frame_col; j < frame_col + 8; ++j) {
48      //Put magnitude into bins based on its corresponding orientation

49        grad_mag=mag_in[i*COLS+j];
50        grad_orien=orien_in[i*COLS+j];
51        char bin0;
52        char bin1;
53        bin0 = (grad_orien - pi18)*pi9_inv;
54        if (grad_orien<pi18)// bin0 will be 0 for orientation < pi18
55          bin0 = -1;
56        bin1 = bin0 + 1;
57        float bin0center = (bin0 * pi9) + pi18;
58        float bin1center = (bin1 * pi9) + pi18;
59        if (bin1>8)
60          bin1=-1;
61        // weight 0 is computed as the factor of the distance from bin1
   center with the bin width
62        float w0 =  (bin1center - grad_orien) *pi9_inv;
63        // weight 1 is computed as the factor of the distance from bin0
   center with the bin width
64        float w1 =  (grad_orien-bin0center) *pi9_inv;
65
66        float temp[9]={0,0,0,0,0,0,0,0,0};
67
68        switch (bin0)
```

```
69          {
70            case -1:
71              temp[0]=grad_mag*w1;
72              break;
73            case 0:
74              temp[0]=grad_mag*w0;
75              temp[1]=grad_mag*w1;
76              break;
77            case 1:
78              temp[1]=grad_mag*w0;
79              temp[2]=grad_mag*w1;
80              break;
81            case 2:
82              temp[2]=grad_mag*w0;
83              temp[3]=grad_mag*w1;
84              break;
85            case 3:
86              temp[3]=grad_mag*w0;
87              temp[4]=grad_mag*w1;
88              break;
89            case 4:
90              temp[4]=grad_mag*w0;
91              temp[5]=grad_mag*w1;
92              break;
93            case 5:
94              temp[5]=grad_mag*w0;
95              temp[6]=grad_mag*w1;
96              break;
97            case 6:
98              temp[6]=grad_mag*w0;
99              temp[7]=grad_mag*w1;
100             break;
101           case 7:
102             temp[7]=grad_mag*w0;
103             temp[8]=grad_mag*w1;
104             break;
105           case 8:
106             temp[8]=grad_mag*w0;
107             break;
108           default:
109             break;
110         }
111
112         shift_bin1[FLOAT_ADD_CYCLES]=shift_bin1[0]+temp[0];
```

```
113         shift_bin2[FLOAT_ADD_CYCLES]=shift_bin2[0]+temp[1];
114         shift_bin3[FLOAT_ADD_CYCLES]=shift_bin3[0]+temp[2];
115         shift_bin4[FLOAT_ADD_CYCLES]=shift_bin4[0]+temp[3];
116         shift_bin5[FLOAT_ADD_CYCLES]=shift_bin5[0]+temp[4];
117         shift_bin6[FLOAT_ADD_CYCLES]=shift_bin6[0]+temp[5];
118         shift_bin7[FLOAT_ADD_CYCLES]=shift_bin7[0]+temp[6];
119         shift_bin8[FLOAT_ADD_CYCLES]=shift_bin8[0]+temp[7];
120         shift_bin9[FLOAT_ADD_CYCLES]=shift_bin9[0]+temp[8];
121                                                 #pragma unroll
122         for (int i=0; i<FLOAT_ADD_CYCLES; i++)
123         {
124           shift_bin1[i]=shift_bin1[i+1];
125           shift_bin2[i]=shift_bin2[i+1];
126           shift_bin3[i]=shift_bin3[i+1];
127           shift_bin4[i]=shift_bin4[i+1];
128           shift_bin5[i]=shift_bin5[i+1];
129           shift_bin6[i]=shift_bin6[i+1];
130           shift_bin7[i]=shift_bin7[i+1];
131           shift_bin8[i]=shift_bin8[i+1];
132           shift_bin9[i]=shift_bin9[i+1];
133         }
134             }
135         }
136 
137     #pragma unroll
138     for(int i=0; i<FLOAT_ADD_CYCLES; i++)
139     {
140       bins[0] += shift_bin1[i];
141     }
142     #pragma unroll
143     for(int i=0; i<FLOAT_ADD_CYCLES; i++)
144     {
145       bins[1] += shift_bin2[i];
146     }
147     #pragma unroll
148     for(int i=0; i<FLOAT_ADD_CYCLES; i++)
149     {
150       bins[2] += shift_bin3[i];
151     }
152     #pragma unroll
153     for(int i=0; i<FLOAT_ADD_CYCLES; i++)
154     {
155       bins[3] += shift_bin4[i];
156     }
```

```
157
158     #pragma unroll
159     for(int i=0; i<FLOAT_ADD_CYCLES; i++)
160     {
161       bins[4] += shift_bin5[i];
162     }
163     #pragma unroll
164     for(int i=0; i<FLOAT_ADD_CYCLES; i++)
165     {
166       bins[5] += shift_bin6[i];
167     }
168     #pragma unroll
169     for(int i=0; i<FLOAT_ADD_CYCLES; i++)
170     {
171       bins[6] += shift_bin7[i];
172     }
173     #pragma unroll
174     for(int i=0; i<FLOAT_ADD_CYCLES; i++)
175     {
176       bins[7] += shift_bin8[i];
177     }
178     #pragma unroll
179     for(int i=0; i<FLOAT_ADD_CYCLES; i++)
180     {
181       bins[8] += shift_bin9[i];
182     }
183
184     #pragma unroll
185     for (int i=0; i< 9; i++){
186       hog_vector_out[count*9+i] = bins[i];
187     }
188
189         count++;
190     }
191 }
```

Listing 5.7 OpenCL code for the Histogram kernel

Normalization kernel

This kernel accesses the global memory to read HOG features, organized in blocks, and write out normalized features. There are different schemes for normalization. L1-sqrt,

calculated by Equation 3.9, is proven to perform equally well compared to L2-norm even though its computation is simpler.

First, a simple version of L1 normalization without square root is implemented. However, the resource usage of this version is too big to fit in the resource available on DE5Net when it is integrated with the other three kernels.

Second, the kernel is optimized by replacing nested loops in the kernel with simple loops as shown in Listing 5.8. The kernel is also implemented with square root function. The comparison between the two version is shown in Table 5.4. Both versions achieve the optimal II. The second version uses less RAM and DSP resource. The number of DSPs is also reduced by 90%.

The kernel's performance is then compared between several platforms. HARP+Arria 10 provides the lowest transfer latency and PACS10 gives the best execution time for this kernel as in Table 5.5.

```
1  #include "../host/inc/defines.h"
2  //Normalization kernel
3  //input: pointer to the hog vector organized in blocks
4  //output: normalized hog vector
5  __kernel
6  void normalization(global const float * restrict hog_vector_in_blocks,
       const int block_num, global float * restrict hog_vector_normalized)
7  {
8      int count = 0;
9      while (count!= block_num)
10     {
11         float L1_sqrt=0;
12         float norm1=0;
13         //do the sum
14         #pragma unroll
15     for (int  j=0; j<NUM_BINS*CELLS_PER_BLOCK; j++)
16     {
17       norm1+=hog_vector_in_blocks[count*NUM_BINS*CELLS_PER_BLOCK+j];
18     }
19
20         //plus an epsilon
21         norm1+=1;
22         //do the division
23     for (int j=0; j<NUM_BINS*CELLS_PER_BLOCK; j++){
24       float tmp=hog_vector_in_blocks[count*NUM_BINS*CELLS_PER_BLOCK+j]/
   norm1;
25       hog_vector_normalized[count*NUM_BINS*CELLS_PER_BLOCK+j]=sqrt(tmp);
26         }
```

```
            count++;
        }
}
```

Listing 5.8 OpenCL code for the Normalization kernel

SVM kernel

The SVM kernel reads the normalized features from global memory and the weight vector obtained by the Support Vectors of the learned model from on-chip memory to generate prediction scores for all windows. The scores are stored back into global memory. This kernel uses on-chip memory to store the weight vector.

This kernel is a latency-critical path of the whole system. The first version, without shift registers, has an initiation interval of up to 289. With shift registers inference, the II of the kernel is just 21. The optimal length of the shift registers is 65. Increasing this length does not improve the initiation interval. The resource overhead is small compared to the improvement of the initiation interval.

```
#define FLOAT_ADD_CYCLES_SVM 65
#include "../host/inc/defines.h"

//SVM prediction kernel
//input: a hog vector window and a weight vector (size 7524)
//output: the prediction value in decimal
__kernel
void svm_predict_values(global const float * restrict weight_vector,
     global const float * restrict hog_vector, const int windows_num,
     global float* restrict dec_values)
{
      float   sum=0;
   local float weight_vector_local[FEATURE_LEN];
   //copy the model into the local variables
   for (int i=0; i<FEATURE_LEN; i++)
   {
      weight_vector_local[i]=weight_vector[i];
   }

   int count = 0;
   while (count!= windows_num)
   {
      int win_row= count/NUM_WINDOWS_X;
      int win_col= count%NUM_WINDOWS_X;
      float   shift_reg[FLOAT_ADD_CYCLES_SVM+1];
```

```
24      #pragma unroll
25      for (int i=0; i<FLOAT_ADD_CYCLES_SVM+1; i++)
26      {
27        shift_reg[i]=0;
28      }
29      #pragma unroll
30      for (int i=win_row; i<win_row+WINDOW_H-1; i++)
31        for (int k=win_col; k<win_col+WINDOW_W-1; k++)
32        {
33          int blk_id=i*NUM_BLOCKS_X+k;
34          int row_in_win=i-win_row;
35          int col_in_win=k-win_col;
36          int blk_id_in_win=row_in_win*(WINDOW_W-1)+col_in_win;
37
38          for (int  j=0; j<NUM_BINS*CELLS_PER_BLOCK_W*CELLS_PER_BLOCK_H; j
    ++)
39          {
40            shift_reg[FLOAT_ADD_CYCLES_SVM]= shift_reg[0] + hog_vector[
    blk_id*CELLS_PER_BLOCK*NUM_BINS+j] *  weight_vector_local[
    blk_id_in_win*CELLS_PER_BLOCK*NUM_BINS+j];
41
42            //shift every element of shift register
43            for (int l=0; l<FLOAT_ADD_CYCLES_SVM; l++)
44            {
45              shift_reg[l] = shift_reg[l+1];
46            }
47          }
48        }
49
50      for (int i=0; i<FLOAT_ADD_CYCLES_SVM; i++)
51      {
52        sum+=shift_reg[i];
53      }
54      dec_values[count++]= sum;
55      sum= 0;
56    }
57
58  }
```

Listing 5.9 OpenCL code for the SVM kernel

Again, the kernel is tested on five platforms. The best execution time is realized on Harp platform but DE5Net gives the best memory transfer time.

Table 5.1 Gradient kernel's resource and performance on different FPGAs

Platform	Fmax (MHz)	ALUTs[1]	FFs	RAMs (20 Kb)	DSPs	II	Execution time (ms)	Transfer time (ms)
OpenVino	142	44,946	40K	46	15	1	21.315	1.857
DE5Net	276	94,160	72K	121	13	1	14.7	0.278
PACS10	323	255,824	270K	173	12	1	1.015	0.002
PAC10	233	128,372	100K	114	15	1	12.088	0.866
HARP	261	214,612	165K	174	14	1	1.377	0.001

[1] Two ALUTs correspond to one ALM..

Table 5.2 Resources usage and performance of the Histogram kernel. The first is the version with fixed-point and line buffer. The second version removes the line buffer. The other versions use float data type. The last version in the Table implements shift registers to reduce loop-carried data dependency.

Version	Fmax (MHz)	ALUTs	FFs	RAMs (20 Kb)	DSPs	II	Execution time (ms)
Fixed-point type, line buffer	240	46,878	49K	111	7	31	NA
Fixed-point type	240	25,646	28K	111	7	31	NA
Float type	240	52,075	49K	162	7	24	NA
Float type, shift registers	246	172,008	152K	176	7	1	16.7

Table 5.3 Histogram kernel's resource and performance on different FPGAs

Platform	Fmax (MHz)	ALUTs	FFs	RAMs (20 Kb)	DSPs	II	Execution time (ms)	Transfer time (ms)
OpenVino	122	127,474	137K	80	7	1	32.646	2.184
DE5Net	246	172,008	152K	176	7	1	16.7	1.168
PACS10	317.35	272,384	300K	188	74	1	65.018	0.761
PAC10	234.08	208,998	163K	152	64	1	16.838	1.536
HARP	222	294,520	278K	285	64	1	27.780	0.937

Table 5.4 Resources usage and performance of the Normalization kernels. One version is L1 normalization with nested loops. The other version is L1-sqrt without nested loops.

Version	Fmax (MHz)	ALUTs	FFs	RAMs	DSPs	II	Execution time (ms)
L1 (nested)	240	75,701	68,496	254	126	1	NA
L1-sqrt	251	136,070	101,378	181	13	1	5.5

Table 5.5 Normalization kernel's resource and performance on different FPGAs

Platform	Fmax (MHz)	ALUTs	FFs	RAMs (20 Kb)	DSPs	II	Execution time (ms)	Transfer time (ms)
OpenVino	121	81,576	72K	90	7	1	11.270	2.106
DE5Net	251	136,070	101K	181	13	1	5.5	0.526
PACS10	298.5	269,054	275K	277	43	1	0.733	1.140
PAC10	249	178,472	124K	157	43	1	6.042	0.967
HARP	227	245,134	178K	431	44	1	1.046	0.425

Table 5.6 Resources usage and performance of the SVM classifier kernels: version with no shift registers and with shift registers

Version	Fmax (MHz)	ALUTs	FFs	RAMs (20 Kb)	DSPs	II	Execution time (ms)
No shift registers	240	68,791	54,833	308	37	289	NA
With shift registers	256	174,752	133,770	335	13	21	53.5

Table 5.7 SVM classifier kernel's resource and performance on different FPGAs

Platform	Fmax (MHz)	ALUTs	FFs	RAMs (20 Kb)	DSPs	II	Execution time (ms)	Transfer time (ms)
OpenVino	112	125,986	130K	267	36	21	193.661	1.093
DE5Net	256	174,752	134K	335	13	21	53.5	0.211
PACS10	268.02	286,798	315K	411	51	21	17.251	0.342
PAC10	240	197,280	140K	315	50	21	24.702	0.546
HARP	221	264,998	204K	436	73	21	23.081	0.386

5.3 Results and comparison

5.3.1 Comparison with multicore CPU

First,the four kernels are compared with their equivalence C/C++ functions running on the host. The host CPU is Intel Core i7. It has 8 cores and runs at 3.5 GHz. Its memory size is 16 GB. Table 5.8 presents the latency of the four functions. The transfer time is not reported in the table because it ranges from 2% to 8% of the execution time, which is small compared to the computation time.

On Intel Core i7 platform, the longest latency components are the gradient and the SVM classifier. Their latency is 10x compared to that of the histogram and the normalization functions. Both these long latency functions are sliding-window tasks. It is confirmed that the sliding-window should be accelerated because it is not software-friendly. FPGAs accelerate the gradient kernel from 124 to 2,363 times depending on the specific platform. The SVM

classifier kernel speeds up from 51 to 163 times. The total latency of the four hardware kernels is from 57x to 102x smaller than that of their corresponding software functions.

5.3.2 Comparison between FPGA platforms

Unlike software functions, the longest latency kernels are the Histogram and the SVM classifier. This is shown in Fig. 5.5. The HARP Xeon+Arria 10 provides the best total latency among the platforms. However, the difference with other platforms is small. Compared to DE5Net and DevCloud PAC10, the HARP Xeon+Arria 10 is slower in the histogram task. Compared to DevCloud PACS10, it is slower in the SVM classifier task.

Table 5.8 Latency comparison with multicore CPU

Platform	Gradient (s)	Histogram (s)	Normalization (s)	SVM classifier (s)	Total (s)
Intel Core i7	2.363	0.116	0.169	2.769	5.417
DE5Net (Stratix V)	0.019	0.017	0.005	0.054	0.095
DevCloud PACS10 (Stratix10)	0.001	0.065	0.001	0.017	0.084
DevCloud PAC10 (Arria10)	0.012	0.017	0.006	0.025	0.06
HARP Xeon+Arria 10	0.001	0.028	0.001	0.023	0.053

Table 5.9 Resources usage and performance on different FPGA platforms

Reference	Platform	Frequenc (MHz)	Image size	LUTs	FFs	RAMs (Kbit)	DSPs	FPS
[150]	Cyclone IV	40	800x600	34,403	23,247	334	68	72
[95]	Virtex-6	150	640x480	184,953	208,666	13,738	190	68.2
[151]	Cyclone IV	50	640x480	6,551	4,375	103	10	72
[146]	Kintex Ulatrascale	NA	800x600	7,804	NA	437	36	115
Ours (HDL)	Cyclone V	50	640x480	13,464	17,117	317	38	11
Ours (HDL)	Arria 10	149	640x480	5,386	2,254	184	2	NA
Ours	DE5Net	241	640x480	311,910	288,829	12,133	65	19
Ours	DevCloud PACS10 (Stratix 10)	230.84	640x480	415,728	466,123	656	180	15
Ours	DevCloud PAC10 (Arria 10)	225.52	640x480	320,026	236,857	11,121	172	40
Ours	HARP Xeon+Arria 10	227.27	640x480	414,760	384,652	16,553	195	36

Table 5.10 Throughput comparison with other heterogeneous platforms

Reference	Platform	Coprocesor	Programming model	Frequency (MHz)	Resolution	FPS
[152]	NVIDIA Quadro 5000	GPU	OpenCL	1,026	768x576	36
[153]	GTX 960	GPU	CUDA	1,178	1,242x375	175
[153]	Tegra X1	GPU	CUDA	1,000	1,242x375	27
Ours	Intel core i7+ DE5Net	FPGA	OpenCL	241	640x480	19
Ours	DevCloud PACS10 (Stratix10)	FPGA	OpenCL	230.84	640x480	15
Ours	DevCloud PAC10 (Arria10)	FPGA	OpenCL	225.52	640x480	40
Ours	HARP Xeon+ Arria 10	FPGA	OpenCL	227.27	640x480	36

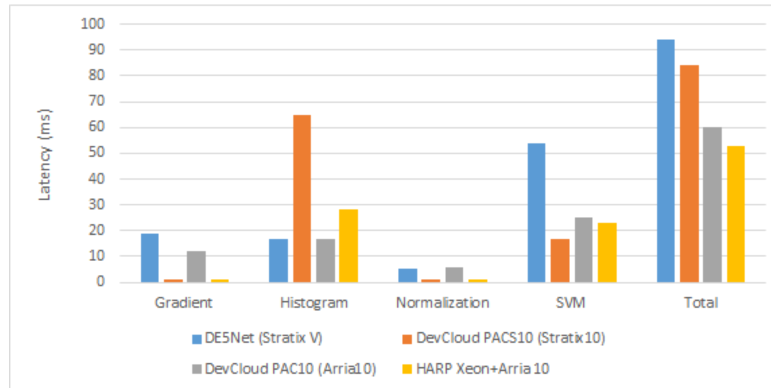


Fig. 5.5 Latency comparison between modules on different FPGAs

5.3.3 FPGA-based pedestrian detection comparison

In the literature, there is not any similar implementation that uses OpenCL to implement HOG+SVM on FPGAs. Therefore, the implementation is compared against HDL-based designs, including our HDL-based design in Chapter 4. Besides, it is compiled onto different FPGA platforms to compare and demonstrate its portability.

From Table 5.9, the OpenCL-based implementations have better FPS than our HDL-based version in Chapter 4. The best performance is on DevCloud PAC10 (Arria 10) with 40 FPS. It is nearly 4× faster than the Verilog-based implementation running at a clock frequency of 50 MHz on Cyclon V. To have a fair comparison, our HDL-based design is also synthesized on Arria 10. It achieves a maximum frequency of 149 MHz which is slower than all the OpenCL-based implementations in Table 5.9. Its throughput is expected to be 33 FPS because the same design is proven to work at 11 FPS with 50 MHz clock frequency on Cyclon V. This throughput is suitable for real-time applications and it is 10 to 20% less than the OpenCL-based implementations on Arria 10 (the last two rows of the Table). The highest FPS is achieved in [146].

Regarding resource usage, OpenCL-based implementations use 10x more than HDL-based ones, except for the DSPs (4.5x). Particularly, in the case of OpenVino Starter Kit, which features a Cyclone V, similar to the DE1-SOC in [154], the combined of the four kernels exceeds the board's available resource. Another comparison is on Arria 10. The HDL-based design uses at least 60x less resource than the OpenCL-based implementation on DevCloud PAC10 (Arria 10).

Finally, it is worth mentioning that OpenCL-based development time is significantly shorter than that of HDL-based approach. The total lines of OpenCL code is 9x smaller as in Table 5.11. The number of Verilog code lines is much bigger in practice because the code

for module instantiation at the top level and the code for communications (I2C, Avalon bus master and slave interfaces) are not taken into account.

Table 5.11 Lines of code comparison between HDL-based and OpenCL-based approach

Module	Gradient	Histogram	Normalization	SVM classifier	Total
HDL	200	789	741	1,305	3,035
OpenCL	48	191	29	58	326

5.3.4 Comparison with other OpenCL-based design

In this section, the implementation is compared with other OpenCL-based designs on GPUs. In [153], the HOG+SVM algorithm is implemented with CUDA. It achieves the state-of-the-art throughput of up to 175 FPS. Using the same implementation, the embedded platform Tegra X1 achieves a real-time speed of 27 FPS. The implementation of this thesis on DevCloud PAC10 is the second to the best with 40 FPS. It is worth noting that DEVCloud PAC10 is not the best platform in terms of latency for separate kernels as shown in Table 5.8. When the kernels are invoked as a pipeline, its throughput is better than the HARP Xeon+Arria10 platform.

5.4 Summary

The chapter presents a HOG+SVM algorithm using OpenCL as a case study of FPGA co-design for ADAS. The design is verified against a golden C model and achieves the equivalent accuracy. Using this approach, the time for implementation and test is greatly reduced. The lines of OpenCL code is only about 10% HDL-based code. It also helps to evaluate and determine the latency bottleneck in the system so that designers can choose which functional module needs to be accelerated. In this case, the latency bottlenecks are in the gradient computing and the SVM classifier module.

Shifting register is the main technique implemented to optimize the latency of the kernels. The target is to minimize the initiation interval of the design as much as possible. The most optimized implementation has a total latency that is from 57 to 102 times smaller compared to that of the software implementation depending on the targeted FPGA platform. The implementation on DevCloud PAC10 achieves the best throughput at 40 FPS, 3 times faster than the Verilog-based implementation in Chapter 4. This comes at a cost of on chip resource. Our experiment result shows that OpenCL implementations on FPGAs work at a higher frequency and throughput yet consumes up to more than 10x amount of resource

compared to HDL-based implementations. Since there is no similar OpenCL implemntation for HOG+SVM on FPGAs, I compare the implementation with OpenCL-based HOG+SVM designs on GPUs. The result show that the GPU-based implementation using CUDA on GTX 960 is up to 4x faster than my current design. In future, it is interesting to port my design onto the same GPU platform to have a closer look at this gap.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

This thesis presents my research on the pedestrian detection problem for autonomous cars. Top accurate algorithms are based on deep neural networks. However, they also demand big amount of computing resources and power consumption. Therefore, I choose to implement a HOG+SVM pipeline to save the resources and power. The pipeline is implemented on FPGAs using two design approaches: RTL and OpenCL coding.

According to my survey in section 2.6, power efficiency of FPGAs is approximately 100 GOPS/W for neural network applications. This number in the case of GPUs and ASICs is 1 TOPS/W and 10 TOPS/W, respectively. However, Table 2.8 shows that FPGA is more energy efficient in some algorithms, including HOG. The reason for this might be the optimization techniques applied for FPGA architecture.

The RTL implementation only uses on-chip memory to store image pixels and intermediate results. It helps avoid accessing external memory which costs more latency and power. In comparison to existing designs on FPGAs, it is the second to the best in terms of number of pixels processed per clock. The design consumes the lowest power. However, in terms of power efficiency measured in FPS per watt, the design is only the third to the best. The design works at a throughput of only 11 FPS, which is exactly the output throughput of the image sensor in the demonstration.

The second implementation of the same system is done using OpenCL programming framework. Using this approach, the design can be synthesized on to different heterogeneous platforms. First, the design is synthesized and compared between different FPGA platforms: DE5Net, DevCloud PACS10, DevCloud PAC10, Harp Xeon. The design is partitioned into four different kernels: gradient, histogram, normalization, and SVM. These kernels on FPGAs speed up the system from 57 to 102 times compared to their software functions. Second, this

implementation is compared against the RTL version. The throughput of OpenCL version is 4x better, yet it uses up to 10x more resources. Finally, it is compared with other OpenCL-based design. To the best of my knowledge, there is not any similar implementation of FPGAs. However, a similar implementation on GPU using CUDA achieves better performance.

6.2 Future directions

One of the experiments for further works is to increase the FPS for the RTL implementation. This is achievable since the current pixel clock frequency for the camera is only 50 Mhz which limits the output pixel throughput of the sensor. Secondly, the design can work at up to 162 MHz frequency, three times of the current setting. Therefore a configuration setting for both the camera and the pipeline to work at 140 MHz is potentially good performance check. The expected throughput for the whole system would be 31 FPS.

Besides, thanks to the OpenCL implementation, the system can be ported to GPU-based systems to see compare the same system on different heterogeneous platforms.

References

[1] *Intel® FPGA SDK for OpenCL™ Pro Edition Programming Guide*, Intel. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683846/21-4/overview.html.

[2] "WHO | Global status report on road safety 2018," Tech. Rep., 2018. [Online]. Available: http://www.who.int/violence_injury_prevention/road_safety_status/2018/en/.

[3] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[4] J. Borrego-Carazo, D. Castells-Rufas, E. Biempica, and J. Carrabina, "Resource-constrained machine learning for adas: A systematic review," *IEEE Access*, vol. 8, pp. 40 573–40 598, 2020. DOI: 10.1109/ACCESS.2020.2976513.

[5] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Understanding the limitations of existing energy-efficient design approaches for deep neural networks," *SysML Conference (SYSML'18)*, vol. 2, no. L1, p. L3, 2018.

[6] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A Survey of FPGA-Based Neural Network Accelerator," *arXiv preprint arXiv:1712.08934*, vol. 9, no. 4, pp. 1–26, 2017. arXiv: 1712.08934. [Online]. Available: http://arxiv.org/abs/1712.08934.

[7] S. Shreejith, K. Vipin, S. A. Fahmy, and M. Lukasiewycz, "An approach for redundancy in flex ray networks using FPGA partial reconfiguration," *Proceedings -Design, Automation and Test in Europe, DATE*, pp. 721–724, 2013, ISSN: 15301591.

[8] J. Ahmad and A. Warren, "FPGA based Deterministic Latency Image Acquisition and Processing System for Automated Driving Systems," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5, ISBN: 9781538648810.

[9] A. M. Lo, A. D. Sappa, and T. Graf, "Survey of Pedestrian Detection for Advanced Driver Assistance Systems," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 7, pp. 1239–1258, 2010.

[10] NHTSA, *Automated Vehicles for Safety*. [Online]. Available: https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety.

[11] J. Janai, F. Güney, A. Behl, A. Geiger, *et al.*, "Computer vision for autonomous vehicles: Problems, datasets and state of the art," *Foundations and Trends® in Computer Graphics and Vision*, vol. 12, no. 1–3, pp. 1–308, 2020.

[12] U. Franke, D. Gavrila, S. Gorzig, F. Lindner, F. Paetzslld, and C. Wohler, "Autonomous driving goes downtown," *IEEE Intelligent Systems and Their Applications*, vol. 13, no. 6, pp. 40–48, 1998.

[13] P. Hurney, “Rmeraeal-time Detection of Pedestrians in Night-time Conditions Using a Vehicle Mounted Infrared Ca,” PhD thesis, National University of Ireland, Galway, 2016.

[14] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dolí, “Microsoft COCO: Common Objects in Context,” in *European conference on computer vision*, 2014, pp. 740–755. arXiv: 1405.0312v3.

[15] H. Alhaija, S. Mustikovela, L. Mescheder, A. Geiger, and C. Rother, “Augmented reality meets computer vision: Efficient data generation for urban driving scenes,” *International Journal of Computer Vision (IJCV)*, 2018.

[16] D. Scharstein and R. Szeliski, “A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms,” *International journal of computer vision*, vol. 47, no. 1-3, pp. 7–42, 2002. [Online]. Available: www.middlebury.edu/stereo..

[17] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge University Press, Mar. 2003. DOI: 10.1017/cbo9780511811685. [Online]. Available: https://www.cambridge.org/core/books/multiple-view-geometry-in-computer-vision/0B6F289C78B2B23F596CAA76D3D43F7A.

[18] R. A. Hamzah and H. Ibrahim, “Literature survey on stereo vision disparity map algorithms,” *Journal of Sensors*, vol. 2016, pp. 1–23, 2016, ISSN: 16877268. DOI: 10.1155/2016/8742920.

[19] H. Hirschmuller and S. Gehrig, “Stereo matching in the presence of sub-pixel calibration errors,” in *IEEE Conference on Computer Vision and Pattern Recognition*, Institute of Electrical and Electronics Engineers (IEEE), Mar. 2009, pp. 437–444. DOI: 10.1109/cvpr.2009.5206493.

[20] N. Smolyanskiy, A. Kamenev, and S. Birchfield, “On the importance of stereo for accurate depth estimation: An efficient semi-supervised deep neural network approach,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 1007–1015.

[21] A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for autonomous driving? the KITTI vision benchmark suite,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3354–3361, ISBN: 9781467312264. DOI: 10.1109/CVPR.2012.6248074.

[22] X. Cheng, Y. Zhong, M. Harandi, Y. Dai, X. Chang, T. Drummond, H. Li, and Z. Ge, “Hierarchical Neural Architecture Search for Deep Stereo Matching,” *Advances in Neural Information Processing Systems*, pp. 1–12, 2020. arXiv: 2010.13501. [Online]. Available: http://arxiv.org/abs/2010.13501.

[23] D. Hernandez-Juarez, A. Chacón, A. Espinosa, D. Vázquez, J. C. Moure, and A. M. López, “Embedded real-time stereo estimation via semi-global matching on the gpu,” *Procedia Computer Science*, vol. 80, pp. 143–153, 2016.

[24] A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for autonomous driving? the kitti vision benchmark suite,” in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, 2012, pp. 3354–3361.

[25] T. Taniai, Y. Matsushita, Y. Sato, and T. Naemura, “Continuous 3d label stereo matching using local expansion moves,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 11, pp. 2725–2739, 2017.

[26] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-Decem, 2016, pp. 3213–3223, ISBN: 9781467388504. DOI: 10.1109/CVPR.2016.350. arXiv: 1604.01685.

[27] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, and A. M. Lopez, "The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3234–3243.

[28] M. Siam, S. Elkerdawy, M. Jagersand, and S. Yogamani, "Deep Semantic Segmentation for Automated Driving : Taxonomy , Roadmap and Challenges," in *2017 IEEE 20th international conference on intelligent transportation systems (ITSC)*, 2017, pp. 1–8. arXiv: arXiv:1707.02432v2.

[29] Ç. Kaymak and A. Uçar, "A Brief Survey and an Application of Semantic Image Segmentation for Autonomous Driving," in *Handbook of Deep Learning Applications*, Springer, 2019, pp. 161–200.

[30] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understandin," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3213–3223. DOI: 10.1080/17843286.1974.11735726.

[31] G. J. Brostow, J. Fauqueur, and R. Cipolla, "Semantic object classes in video: A high-definition ground truth database," *Pattern Recognition Letters*, vol. 30, no. 2, pp. 88–97, 2009, ISSN: 01678655. DOI: 10.1016/j.patrec.2008.04.005.

[32] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*, 2015, pp. 234–241.

[33] J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Adaptation Networks for Semantic Segmentation," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 3431–3440, ISBN: 9781538664209. DOI: 10.1109/CVPR.2018.00712. arXiv: 1804.08286.

[34] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[35] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid scene parsing network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2881–2890, ISBN: 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.660. arXiv: arXiv:1606.00915. [Online]. Available: https://github.com/hszhao/PSPNet.

[36] Y. Zhu, K. Sapra, F. A. Reda, K. J. Shih, S. Newsam, A. Tao, and B. Catanzaro, "Improving semantic segmentation via video propagation and label relaxation," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2019-June, pp. 8848–8857, 2019, ISSN: 10636919. DOI: 10.1109/CVPR.2019.00906. arXiv: 1812.01593.

[37] B. Zoph, G. Ghiasi, T.-Y. Lin, Y. Cui, H. Liu, E. D. Cubuk, and Q. V. Le, "Rethinking pre-training and self-training," *arXiv preprint arXiv:2006.06882*, 2020.

[38] G. J. Brostow, J. Fauqueur, and R. Cipolla, "Semantic object classes in video: A high-definition ground truth database," *Pattern Recognition Letters*, vol. 30, no. 2, pp. 88–97, 2009.

[39] A. Tao, K. Sapra, and B. Catanzaro, "Hierarchical Multi-Scale Attention for Semantic Segmentation," pp. 1–11, 2020. arXiv: 2005.10821. [Online]. Available: http://arxiv.org/abs/2005.10821.

[40] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *IEEE international conference on computer vision*, 2017, pp. 2961–2969. arXiv: 1703.06870v3. [Online]. Available: https://github.com/.

[41] M. Everingham, S. M. Eslami, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes Challenge: A Retrospective," *International Journal of Computer Vision*, vol. 111, no. 1, pp. 98–136, 2015, ISSN: 15731405. DOI: 10.1007/s11263-014-0733-5.

[42] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2014, pp. 580–587, ISBN: 9781479951178. DOI: 10.1109/CVPR.2014.81. arXiv: 1311.2524.

[43] X. Feng, Y. Jiang, X. Yang, M. Du, and X. Li, "Computer vision algorithms and hardware implementations: A survey," *Integration*, vol. 69, no. July, pp. 309–320, 2019, ISSN: 01679260. DOI: 10.1016/j.vlsi.2019.07.005. [Online]. Available: https://doi.org/10.1016/j.vlsi.2019.07.005.

[44] L. Jiao, F. Zhang, F. Liu, S. Member, S. Yang, L. Li, Z. Feng, and R. Qu, "A Survey of Deep Learning-based Object Detection," *IEEE Access*, vol. 7, pp. 128 837–128 868, 2019. arXiv: 1907.09408v2.

[45] H. Wang, Y. Yu, Y. Cai, X. Chen, L. Chen, and Q. Liu, "A Comparative Study of State-of-the-Art Deep Learning Algorithms for Vehicle Detection," *IEEE Intelligent Transportation Systems Magazine*, vol. 11, no. 2, pp. 82–95, 2019, ISSN: 19411197. DOI: 10.1109/MITS.2019.2903518.

[46] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature Pyramid Networks for Object Detection," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 2117–2125.

[47] A. Bar Hillel, R. Lerner, D. Levi, and G. Raz, "Recent progress in road and lane detection: A survey," *Machine Vision and Applications*, vol. 25, no. 3, pp. 727–745, 2014, ISSN: 14321769. DOI: 10.1007/s00138-011-0404-2.

[48] Chao Fan, Y.-P. Song, and J. Ya-Jie, "Multi-Lane Detection Based on Deep Convolutional Neural Network," *IEEE Access*, vol. 7, pp. 150 833–150 841, 2019.

[49] *TuSimple dataset*. [Online]. Available: https://github.com/TuSimple/tusimple-benchmark/wiki.

[50] Z. Qu, H. Jin, Y. Zhou, Z. Yang, and W. Zhang, "Focus on Local: Detecting Lane Marker from Bottom Up via Key Point," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 14 122–14 130. arXiv: 2105.13680. [Online]. Available: http://arxiv.org/abs/2105.13680.

[51] X. Pan, J. Shi, P. Luo, X. Wang, and X. Tang, "Spatial as deep: Spatial CNN for traffic scene understanding," *Procedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018. arXiv: 1712.06080.

[52] L. Liu, X. Chen, S. Zhu, and P. Tan, "Condlanenet: A top-to-down lane detection framework based on conditional convolution," *arXiv preprint arXiv:2105.05003*, 2021.

[53] K. Behrendt and R. Soussan, "Unsupervised labeled lane markers using maps," in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, 2019.

[54] F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhavan, and T. Darrell, "Bdd100k: A diverse driving dataset for heterogeneous multitask learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 2636–2645.

[55] M. Lee, J. Lee, D. Lee, W. Kim, S. Hwang, and S. Lee, "Robust Lane Detection via Expanded Self Attention," 2021. arXiv: 2102.07037. [Online]. Available: http://arxiv.org/abs/2102.07037.

[56] D. Neven, B. De Brabandere, S. Georgoulis, M. Proesmans, and L. Van Gool, "Towards End-to-End Lane Detection: An Instance Segmentation Approach," *IEEE Intelligent Vehicles Symposium, Proceedings*, vol. 2018-June, pp. 286–291, 2018. DOI: 10.1109/IVS.2018.8500547. arXiv: 1802.05591.

[57] P. Phalguni, K. Ganapathi, V. Madumbu, R. Rajendran, and S. David, "Design and implementation of an automatic traffic sign recognition system on TI OMAP-L138," *2013 IEEE International Conference on Industrial Technology (ICIT)*, pp. 1104–1109, 2013. DOI: 10.1109/ICIT.2013.6505826.

[58] A. Mogelmose, M. M. Trivedi, and T. B. Moeslund, "Vision based Traffic Sign Detection and Analysis for Intelligent Driver Assistance Systems: Perspectives and Survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 13, no. 4, pp. 1484–1497, 2012.

[59] S. B. Wali, M. A. Abdullah, M. A. Hannan, A. Hussain, S. A. Samad, P. J. Ker, and M. B. Mansor, *Vision-based traffic sign detection and recognition systems: Current trends and challenges*, May 2019. DOI: 10.3390/s19092093.

[60] M. Mathias, R. Timofte, R. Benenson, and L. Van Gool, "Traffic sign recognition - How far are we from the solution?" In *Proceedings of the International Joint Conference on Neural Networks*, 2013, ISBN: 9781467361293. DOI: 10.1109/IJCNN.2013.6707049.

[61] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Networks*, vol. 32, pp. 323–332, Aug. 2012, ISSN: 08936080. DOI: 10.1016/j.neunet.2012.02.016.

[62] S. Houben, J. Stallkamp, J. Salmen, M. Schlipsing, and C. Igel, "Detection of traffic signs in real-world images: The German traffic sign detection benchmark," in *Proceedings of the International Joint Conference on Neural Networks*, 2013, ISBN: 9781467361293. DOI: 10.1109/IJCNN.2013.6706807.

[63] D. Tabernik and D. Skocaj, "Deep Learning for Large-Scale Traffic-Sign Detection and Recognition," *IEEE Transactions on Intelligent Transportation Systems*, vol. PP, pp. 1–14, 2019, ISSN: 1524-9050. DOI: 10.1109/tits.2019.2913588. arXiv: 1904.00649.

[64] C. Gamez Serna and Y. Ruichek, "Traffic Signs Detection and Classification for European Urban Environments," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 10, pp. 4388–4399, 2020, ISSN: 15580016. DOI: 10.1109/TITS.2019.2941081.

[65] M. B. Jensen, M. P. Philipsen, A. Møgelmose, T. B. Moeslund, and M. M. Trivedi, "Vision for Looking at Traffic Lights: Issues, Survey, and Perspectives," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 7, pp. 1800–1815, Jul. 2016, ISSN: 15249050. DOI: 10.1109/TITS.2015.2509509.

[66] M. Diaz, P. Cerri, G. Pirlo, M. A. Ferrer, and D. Impedovo, "A survey on traffic light detection," in *International Conference on Image Analysis and Processing*, 2015, pp. 201–208, ISBN: 9783319232218. DOI: 10.1007/978-3-319-23222-5_25.

[67] M. Bach, D. Stumper, and K. Dietmayer, "Deep Convolutional Traffic Light Recognition for Automated Driving," in *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, Institute of Electrical and Electronics Engineers Inc., Dec. 2018, pp. 851–858, ISBN: 9781728103235. DOI: 10.1109/ITSC.2018.8569522.

[68] *Lara dataset*, Accessed: 2021-06-03, 2010. [Online]. Available: http://www.lara.prd.fr/benchmarks/trafficlightsrecognition.

[69] S.-c. Lin, Y. Zhang, C.-h. Hsu, M. Skach, E. Haque, L. Tang, and J. Mars, "The Architectural Implications of Autonomous Driving : Constraints and Acceleration," *Proce. ASPLOS'18*, pp. 751–766, 2018. DOI: 10.1145/3173162.3173191.

[70] H. K. Kim, K. Y. Yoo, J. H. Park, and H. Y. Jung, "Traffic light recognition based on binary semantic segmentation network," *Sensors (Switzerland)*, vol. 19, no. 7, pp. 1–15, 2019, ISSN: 14248220. DOI: 10.3390/s19071700.

[71] M. P. Philipsen, M. B. Jensen, A. Mogelmose, T. Moseslund, and M. M. Trivedi, "Learning based traffic light detection: Evaluation on challenging dataset," in *18th IEEE Intelligent Transportation Systems Conference*, 2015.

[72] K. Behrendt and L. Novak, "A deep learning approach to traffic lights: Detection, tracking, and classification," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, IEEE.

[73] A. Fregin, J. Muller, U. Krebel, and K. Dietmayer, "The driveu traffic light dataset: Introduction and comparison with existing datasets," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 3376–3383. DOI: 10.1109/ICRA.2018.8460737.

[74] J. Muller and K. Dietmayer, "Detecting Traffic Lights by Single Shot Detection," *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, vol. 2018-Novem, pp. 266–273, 2018. DOI: 10.1109/ITSC.2018.8569683. arXiv: 1805.02523.

[75] N. Bernini, M. Bertozzi, L. Castangia, M. Patander, and M. Sabbatelli, "Real-time obstacle detection using stereo vision for autonomous ground vehicles: A survey," in *17th IEEE International Conference on Intelligent Transportation Systems, ITSC 2014*, 2014, pp. 873–878, ISBN: 9781479960781. DOI: 10.1109/ITSC.2014.6957799. arXiv: 1204.3968.

[76] V. D. Nguyen, T. T. Nguyen, D. D. Nguyen, S. J. Lee, and J. W. Jeon, "A fast evolutionary algorithm for real-time vehicle detection," *IEEE Transactions on Vehicular Technology*, vol. 62, no. 6, pp. 2453–2468, 2013, ISSN: 00189545. DOI: 10.1109/TVT.2013.2242910.

[77] Hernan Badino, Uwe Franke and D. Pfeiffer, "The Stixel World - A Compact Medium Level Representation of the 3D-World," in *Pattern recognition 31st DAGM symposium*, 2009, pp. 51–60, ISBN: 978-3-540-22945-2. DOI: 10.1088/0305-4624/8/5/I01. arXiv: arXiv:1011.1669v3.

[78] R. M. Hernan Badino, Uwe Franke, "Free Space Computation Using Stochastic Occupancy Grids and Dynamic Programming," in *International conference on Computer Vision, Workshop Dynamical Vision*, 2007.

[79] F. U., R. C., B. H., and G. S., "6D-Vision: Fusion of Stereo and Motion for Robust Environment Perception," in *Joint Pattern Recognition Symposium. Springer, Berlin, Heidelberg,*, 2005, pp. 216–223, ISBN: 3540287035. DOI: 10.1007/11550518_27.

[80] D. Pfeiffer and U. Franke, "Efficient representation of traffic scenes by means of dynamic stixels," *IEEE Intelligent Vehicles Symposium, Proceedings*, pp. 217–224, 2010, ISSN: 1931-0587. DOI: 10.1109/IVS.2010.5548114.

[81] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal Loss for Dense Object Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 2, pp. 318–327, Aug. 2017. arXiv: 1708.02002. [Online]. Available: http://arxiv.org/abs/1708.02002.

[82] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9905 LNCS, pp. 21–37, 2016, ISSN: 16113349. DOI: 10.1007/978-3-319-46448-0_2. arXiv: 1512.02325.

[83] L. Sun, K. Yang, X. Hu, W. Hu, and K. Wang, "Real-Time fusion network for rgb-d sementic segmentation incorporating unexpected obstacle detection," *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 5558–5565, 2020, ISSN: 23773766. DOI: 10.1109/LRA.2020.3007457. arXiv: arXiv:2002.10570v2.

[84] P. Pinggera, S. Ramos, S. Gehrig, U. Franke, C. Rother, and R. Mester, "Lost and found: Detecting small road hazards for self-driving vehicles," in *IEEE International Conference on Intelligent Robots and Systems*, vol. 2016-Novem, 2016, pp. 1099–1106, ISBN: 9781509037629. DOI: 10.1109/IROS.2016.7759186. arXiv: 1609.04653.

[85] A. DeHon, "Trends toward spatial computing architectures," in *1999 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC. First Edition (Cat. No.99CH36278)*, 1999, pp. 362–363. DOI: 10.1109/ISSCC.1999.759296.

[86] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[87] D. Castells-Rufas, A. Saa-Garriga, and J. Carrabina, "Energy efficiency of many-soft-core processors," *arXiv preprint arXiv:1601.07133*, 2016.

[88] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 5–14, ISBN: 9781450343541. DOI: 10.1145/3020078.3021740. [Online]. Available: https://doi.org/10.1145/3020078.3021740.

[89] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016. DOI: 10.1109/TCAD.2015.2513673.

[90] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017, ISSN: 00189219. DOI: 10.1109/JPROC.2017.2761740. arXiv: 1703.09039.

[91] Y. Wang, S. Liang, S. Yao, Y. Shan, S. Han, J. Peng, and H. Luo, "RECONFIGURABLE PROCESSOR FOR DEEP LEARNING IN AUTONOMOUS VEHICLES," Tech. Rep. [Online]. Available: https://www.itu.int/en/journal/001/Pages/default.aspx.

[92] K. Matsubara, L. Hanno, M. Kimura, A. Nakamura, M. Koike, K. Terashima, S. Morikawa, Y. Hotta, T. Irita, S. Mochizuki, H. Hamasaki, and T. Kamei, "4.2 A 12nm Autonomous-Driving Processor with 60.4TOPS, 13.8TOPS/W CNN Executed by Task-Separated ASIL D Control," in *IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 56–58, ISBN: 9781728195490. DOI: 10.1109/ISSCC42613.2021.9365745.

[93] W. Shi, X. Li, Z. Yu, and G. Overett, "An FPGA-Based Hardware Accelerator for Traffic Sign Detection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 4, pp. 1362–1372, 2017, ISSN: 10638210. DOI: 10.1109/TVLSI.2016.2631428.

[94] O. Rahnama, T. Cavalleri, S. Golodetz, S. Walker, and P. Torr, "R3SGM : Real-time Raster-Respecting Semi-Global Matching for Power-Constrained Systems," in *International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 102–109, ISBN: 9781728102146. DOI: 10.1109/FPT.2018.00025.

[95] X. Ma, W. A. Najjar, and A. K. Roy-Chowdhury, "Evaluation and acceleration of high-throughput fixed-point object detection on FPGAS," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 6, pp. 1051–1062, 2015, ISSN: 10518215. DOI: 10.1109/TCSVT.2014.2360030.

[96] H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato, "A Lightweight YOLOv2: A Binarized CNN with A Parallel Support Vector Regression for an FPGA," *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*, pp. 31–40, 2018. DOI: 10.1145/3174243.3174266. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3174243.3174266.

[97] G. Chen, Y. Ling, T. He, H. Meng, S. He, Y. Zhang, and K. Huang, “StereoEngine: An FPGA-Based Accelerator for Real-Time High-Quality Stereo Estimation with Binary Neural Network,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4179–4190, Nov. 2020, ISSN: 19374151. DOI: 10.1109/TCAD.2020.3012864. [Online]. Available: https://ieeexplore.ieee.org/document/9211569/.

[98] R. Retting, “Traffic Fatalities by State,” Governors Highway Safety Association, Tech. Rep., 2020.

[99] P. Dollár, C. Wojek, B. Schiele, and P. Perona, “Pedestrian Detection: A Benchmark,” in *Proc. CVPR*, 2009, pp. 304–311, ISBN: 9781424439911. DOI: 10.1109/CVPR.2009.5206631.

[100] N. Dalal and W. Triggs, “Histograms of Oriented Gradients for Human Detection,” *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition CVPR05*, vol. 1, no. 3, pp. 886–893, 2004. DOI: 10.1109/CVPR.2005.177.

[101] M. Enzweiler and D. M. Gavrila, “Monocular pedestrian detection: Survey and experiments,” in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, 2009, pp. 2179–2195. DOI: 10.1109/TPAMI.2008.260.

[102] A. Ess, B. Leibe, and L. Van Gool, “Depth and appearance for mobile scene analysis,” in *2007 IEEE 11th international conference on computer vision*, IEEE, 2007, pp. 1–8.

[103] S. W. C. Wojek and B. Schiele, “Multi-Cue Onboard Pedestrian Detection,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 794–801.

[104] S. Zhang, R. Benenson, and B. Schiele, “CityPersons: A Diverse Dataset for Pedestrian Detection,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 3213–3221. arXiv: arXiv:1702.05693v1.

[105] M. Braun, S. Krebs, F. Flohr, and D. M. Gavrila, “Eurocity persons: A novel benchmark for person detection in traffic scenes,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 8, pp. 1844–1861, 2019.

[106] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, “The pascal visual object classes (voc) challenge,” *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[107] P. Dollár, C. Wojek, B. Schiele, and P. Perona, “Pedestrian detection: An evaluation of the state of the art,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 4, pp. 743–761, 2012, ISSN: 01628828. DOI: 10.1109/TPAMI.2011.155.

[108] J. Cao, Y. Pang, J. Xie, F. S. Khan, and L. Shao, “From handcrafted to deep features for pedestrian detection: A survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2021. DOI: 10.1109/TPAMI.2021.3076733.

[109] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004. DOI: 10.1023/B:VISI.0000029664.99615.94.

[110] T. Ojala, M. Pietikainen, and T. Maenpaa, “Multiresolution gray-scale and rotation invariant texture classification with local binary patterns,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 7, pp. 971–987, 2002. DOI: 10.1109/TPAMI.2002.1017623.

[111] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417, ISBN: 978-3-540-33833-8.

[112] P. Viola and M. J. Jones, "Robust real-time face detection," *International Journal of Computer Vision*, vol. 57, pp. 137–154, 2004. DOI: 10.1023/B:VISI.0000013087.49260.fb.

[113] ——, "Detecting Pedestrians Using Patterns of Motion and Appearance," *International Journal of Computer Vision*, vol. 63, no. 2, pp. 153–161, 2005.

[114] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[115] R. E. Schapire and Y. Singer, "Improved boosting algorithms using confidence-rated predictions," *Machine learning*, vol. 37, no. 3, pp. 297–336, 1999.

[116] *Caltech Pedestrian Detection Benchmark*, 2018. [Online]. Available: http://www.vision.caltech.edu/Image_Datasets/CaltechPedestrians/.

[117] I. Hasan, S. Liao, J. Li, S. U. Akram, and L. Shao, "Generalizable pedestrian detection: The elephant in the room," *arXiv preprint arXiv:2003.08799*, vol. 1, no. 2, 2020.

[118] P. Viola and M. J. Jones, "Robust real-time face detection," *International journal of computer vision*, vol. 57, no. 2, pp. 137–154, 2004.

[119] Z. Cai, Q. Fan, R. S. Feris, and N. Vasconcelos, "A unified multi-scale deep convolutional neural network for fast object detection," in *European conference on computer vision*, Springer, 2016, pp. 354–370.

[120] J. Li, X. Liang, S. Shen, T. Xu, J. Feng, and S. Yan, "Scale-aware fast r-cnn for pedestrian detection," *IEEE transactions on Multimedia*, vol. 20, no. 4, pp. 985–996, 2017.

[121] L. Zhang, L. Lin, X. Liang, and K. He, "Is faster r-cnn doing well for pedestrian detection?" In *European conference on computer vision*, Springer, 2016, pp. 443–457.

[122] X. Du, M. El-Khamy, J. Lee, and L. Davis, "Fused DNN: A deep neural network fusion approach to fast and robust pedestrian detection," *Proceedings - 2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017*, pp. 953–961, 2017. DOI: 10.1109/WACV.2017.111. arXiv: 1610.03466.

[123] X. Zhang, L. Cheng, B. Li, and H.-M. Hu, "Too far to see? not really!—pedestrian detection with scale-aware localization policy," *IEEE transactions on image processing*, vol. 27, no. 8, pp. 3703–3715, 2018.

[124] S. Wang, J. Cheng, H. Liu, and M. Tang, "Pcn: Part and context information for pedestrian detection with cnns," *arXiv preprint arXiv:1804.04483*, 2018.

[125] C. Lin, J. Lu, G. Wang, and J. Zhou, "Graininess-aware deep feature learning for pedestrian detection," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 732–747.

[126] X. Du, M. El-Khamy, V. I. Morariu, J. Lee, and L. Davis, "Fused Deep Neural Networks for Efficient Pedestrian Detection," pp. 1–11, 2018. arXiv: 1805.08688. [Online]. Available: http://arxiv.org/abs/1805.08688.

[127] T. Song, L. Sun, D. Xie, H. Sun, and S. Pu, "Small-scale pedestrian detection based on somatic topology localization and temporal feature aggregation," *arXiv preprint arXiv:1807.01438*, 2018.

[128] G. Brazil, X. Yin, and X. Liu, "Illuminating Pedestrians via Simultaneous Detection and Segmentation," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-Octob, pp. 4960–4969, 2017, ISSN: 15505499. DOI: 10.1109/ICCV.2017.530. arXiv: 1706.08564.

[129] G. Brazil and X. Liu, "Pedestrian detection with autoregressive network phases," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2019-June, pp. 7224–7233, 2019, ISSN: 10636919. DOI: 10.1109/CVPR.2019.00740. arXiv: 1812.00440.

[130] A. H. Khan, M. Munir, L. van Elst, and A. Dengel, *F2dnet: Fast focal detection network for pedestrian detection*, 2022. DOI: 10.48550/ARXIV.2203.02331. [Online]. Available: https://arxiv.org/abs/2203.02331.

[131] W. Liu, S. Liao, W. Ren, W. Hu, and Y. Yu, "High-level semantic feature detection: A new perspective for pedestrian detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 5187–5196.

[132] I. Hasan, S. Liao, J. Li, S. U. Akram, and L. Shao, "Pedestrian detection: The elephant in the room," *arXiv preprint arXiv:2003.08799*, 2020.

[133] N. Dalal, F. People, and V. H.-c. Interaction, "Finding People in Images and Videos Navneet Dalal To cite this version : HAL Id : tel-00390303," 2009.

[134] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995, ISSN: 0885-6125. DOI: 10.1007/bf00994018.

[135] T. Joachims, "Text Classification," in *Learning to Classify Text Using Support Vector Machines*, Springer US, 2002, pp. 7–33. DOI: 10.1007/978-1-4615-0907-3_2.

[136] C.-J. Chang, Chih-Chung and Lin, "{LIBSVM}: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, 27:1–27:27, 2011.

[137] *DE1-SoC Board*. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836.

[138] *The 5 Mega Pixel Digital Camera Development Package*. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=68&No=281.

[139] D. Castells, *PedestrianDetectionBenchmark*, 2019. [Online]. Available: https://github.com/vinhphuong1501/hog%7B%5C_%7Dbenchmark.

[140] J. Rettkowski, A. Boutros, and D. Göhringer, "Real-time pedestrian detection on a xilinx zynq using the HOG algorithm," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, IEEE, Dec. 2015, pp. 1–8, ISBN: 978-1-4673-9406-2. DOI: 10.1109/ReConFig.2015.7393339. [Online]. Available: http://ieeexplore.ieee.org/document/7393339/.

[141] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "FPGA-Based real-time pedestrian detection on high-resolution images," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 629–635, 2013, ISSN: 21607508. DOI: 10.1109/CVPRW.2013.95.

[142] S. Ghaffari, P. Soleimani, K. F. Li, and D. W. Capson, "A Novel Hardware–Software Co-Design and Implementation of the HOG Algorithm," *Sensors*, vol. 20, no. 19, p. 5655, Oct. 2020, ISSN: 1424-8220. DOI: 10.3390/s20195655. [Online]. Available: https://www.mdpi.com/1424-8220/20/19/5655.

[143] A. Khan, M. U. K. Khan, M. Bilal, and C. M. Kyung, "Hardware architecture and optimization of sliding window based pedestrian detection on FPGA for high resolution images by varying local features," *IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC*, vol. 2015-Octob, pp. 142–148, 2015, ISSN: 23248440. DOI: 10.1109/VLSI-SoC.2015.7314406.

[144] C. Blair, N. M. Robertson, and D. Hume, "Characterising a Heterogeneous System for Person Detection in Video using Histograms of Oriented Gradients: Power vs. Speed vs. Accuracy," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, no. 2, pp. 236–247, 2013, ISSN: 2156-3357. DOI: 10.1109/JETCAS.2013.2256821.

[145] J. H. Luo and C. H. Lin, "Pure FPGA Implementation of an HOG Based Real-Time Pedestrian Detection System," *Sensors*, vol. 18, no. 4, 2018, ISSN: 1424-8220. DOI: 10.3390/s18041174. [Online]. Available: https://www.mdpi.com/1424-8220/18/4/1174.

[146] S. Ghaffari, P. Soleimani, K. F. Li, and D. W. Capson, "A Novel Hardware–Software Co-Design and Implementation of the HOG Algorithm," *Sensors*, vol. 20, no. 19, p. 5655, Oct. 2020, ISSN: 1424-8220. DOI: 10.3390/s20195655. [Online]. Available: https://www.mdpi.com/1424-8220/20/19/5655.

[147] *IntelFPGA SDK for OpenCL Pro edition: Best Practice Guide*, English, Intel, 194 pp.

[148] *SDAccel environment profiling and optimization guide*, English, version V2017.4, Xilinx, 99 pp.

[149] H. R. Zohouri, *High performance computing with fpgas and opencl*, 2019. arXiv: 1810.09773 [cs.DC].

[150] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, and M. Yoshimoto, "Architectural study of HOG feature extraction processor for real-time object detection," *IEEE Workshop on Signal Processing Systems, SiPS: Design and Implementation*, pp. 197–202, 2012, ISSN: 15206130. DOI: 10.1109/SiPS.2012.57.

[151] M. Bilal, A. Khan, M. Umar, K. Khan, and C.-m. Kyung, "A Low-Complexity Pedestrian Detection Framework for Smart Video Surveillance Systems," vol. 27, no. 10, pp. 2260–2273, 2017.

[152] R. Sun, X. Wang, and X. Ye, "Real-time pedestrian detection Using OpenCL," in *ICALIP 2014 - 2014 International Conference on Audio, Language and Image Processing, Proceedings*, Institute of Electrical and Electronics Engineers Inc., Jan. 2015, pp. 401–404, ISBN: 9781479939022. DOI: 10.1109/ICALIP.2014.7009824.

[153] V. Campmany, S. Silva, A. Espinosa, J. C. Moure, D. Vázquez, and A. M. López, "GPU-based pedestrian detection for autonomous driving," in *Procedia Computer Science*, vol. 80, Elsevier B.V., 2016, pp. 2377–2381. DOI: 10.1016/j.procs.2016.05.455. arXiv: 1611.01642.

[154] V. Ngo, D. Castells-Rufas, A. Casadevall, M. Codina, and J. Carrabina, "Low-Power Pedestrian Detection System on FPGA," in *Multidisciplinary Digital Publishing Institute Proceedings*, vol. 31, Nov. 2019, p. 35. DOI: 10.3390/proceedings2019031035. [Online]. Available: https://www.mdpi.com/2504-3900/31/1/35.

Appendix A

Calculate the FPS of output images from the sensor

A.1 Camera settings

Some key configurations for the image sensor [138] used in this thesis are listed in Table A.1.

Table A.1 Pixel clock settings

Parameter	Value	Comments
XCLKIN	25 MHz	Clock input to the sensor, must be in the range from 6 Mhz to 27 MHz
PLL_m_Factor	24	M = PLL_m_Factor
PLL_n_Divider	5	N = PLL_n_Divider + 1
PLL_p1_Divider	1	P1 = PLL_p1_Divider + 1
Column size	2559	The width of the field of view in pixels
Row size	1919	The height of the field of view in pixels
Column Skip	1	The width of the field of view is reduced 2X
Column Bin	1	Set in conjunction with skipping. One neighbor pixel in the same row is averaged with each output
Row Skip	1	The height of the field of view is reduced 2X
Row Bin	1	Set in conjunction with skipping. One neighbor pixel in the same col. is averaged with each output
Shutter Width Upper	0	Exposure time (in number of row time) upper bytes
Shutter Width Lower	1984	Exposure time (in number of row time) lower bytes
Horizontal Blank	0	Horizontal blanking in pixel clocks
Vertical Blank	25	Vertical blanking in pixel clocks

The pixel clock output of the camera is calculated as in Equation A.1. Accordingly, the camera outputs raw pixels at 50 MHz.

$$PIXCLK = (XCLKIN \times M)/(N \times P1) \tag{A.1}$$

A.2 Frame rate calculation

The Table A.2 presents the parameters and their corresponding values to calculate the frame rate at the input of the pedestrian detection system. The pixel clock, which is 50 MHz, is calculated from section A.1. According to [138], the WDC (Dark columns after binning) parameter is 40 because the Column Bin is set to 1 as in Table A.1. According to the calculations in Table A.2, the frame rate at the output of the image sensor is 11 FPS.

Table A.2 Frame rate calculation

Parameter	Name	Equation	Result
FPS	Frame rate	$1/tFrame$	11
tFrame	Frame time	$(H + max(VB, VBMIN)) \times tROW$	$89ms$
tROW	Row time	$2 \times tPIXCLK \times max(((W/2) + max(HB, HBMIN)),$ $(41 + 208 \times (Row_Bin + 1) + 99))$	$45ms$
W	Image Width	$2 \times ceil((Column_Size + 1)/(2 \times (Column_Skip + 1)))$	1280
H	Image Height	$2 \times ceil((Row_Size + 1)/(2 \times (Row_Skip + 1)))$	960
SW	Shutter Width	$max(1, (2^{16} \times Shutter_Width_Upper)$ $+Shutter_Width_Lower)$	1984
HB	Horizontal Blanking	$Horizontal_Blank + 1$	1
VB	Vertical Blanking	$Vertical_Blank + 1$	26
HBMIN	Minimum Horizontal Blanking	$208 \times (Row_Bin + 1) + 64 + (WDC/2)$	500
VBMIN	Minimum Vertical Blanking	$max(8, SW - H) + 1$	1025
tPIXCLK	Pixel Clock	$1/fPIXCLK$	$20ns$