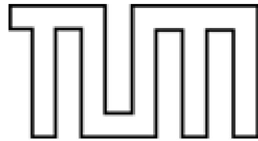


FAKULTÄT FÜR INFORMATIK

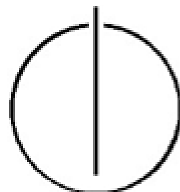
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

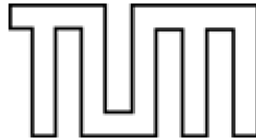
Master's Thesis in Informatics

Lane Detection and Tracking for Advanced Driver Assistant System

Multiple regions of interest and performance evaluation for the lane detection and tracking in a heterogeneous platform

Alibi Rakhmatulin





FAKULTÄT FÜR INFORMATIK

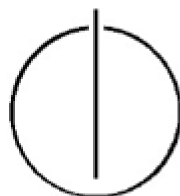
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Multiple regions of interest and performance evaluation for the lane detection and tracking in a heterogeneous platform

Multi ROI und Leistungsbewertung für die Fahrspur Erkennung und -verfolgung in einer heterogenen Plattform

Author:	Alibi Rakhmatulin
Supervisor:	Prof. Dr.-Ing. habil. Alois Knoll
Advisors:	Dr. Kai Huang, Ph.D
	Biao Hu, M.Sc
Date:	May 15, 2016



I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

Munich, Germany   Alibi Rakhmatulin

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor Dr. Kai Huang for giving me an opportunity to write the thesis at the chair of Robotics and Embedded Systems. I thank him for all the support and guidance that he has given me in the course HW-SW Co-design and this thesis.

I would also sincerely thank my advisor, Mr. Biao Hu, without his support the thesis would not have been successful. He has guided me through each step of the entire thesis and his valuable suggestions and insights into the topic made the work easier than it would have been. Our discussions were as enjoyable as they were productive and I was always able to have a meeting at any time without prior appointments, he was always available, even during holidays.

Abstract

Previous Lane detection and Tracking algorithm suffers from low performance due to partial execution on high performance heterogeneous hardware. Significant amount of image processing tasks meant to be computed by hardware accelerators are assigned to CPUs.

In this thesis maximum workload related to image processing was delegated to hardware accelerators by developing Open CL kernel. This allowed to harvest parallel processing capabilities of GPUs and FPGAs thus moving away from ECUs and improving the performance. For instance, pre-processing of an image was completely transferred to hardware accelerators. Besides algorithm supporting independent multiple self-adjustable regions of interest was developed, programming bottlenecks were eliminated, lane inclination angle across all the regions of interest is calculated and performance evaluation is performed on independent recorded videos and benchmarking datasets.

Lane Detection and tracking algorithm is based on Particle Filter. In Lane detection phase pre-processed image is populated with thousands of randomly placed sample lines, weights of those lines are calculated by hardware accelerators and fittest line is selected to represent the actual road lane marking. Lane tracking is based on Particle Filter and weighing obtained from Lane Detection phase.

Algorithm was developed and tested on the following hardware GPU: Nvidia GeForce GTX 660 TI, Altera Cyclone V and RAM 32 GB, Intel Core i8 and a set of pre-recorded videos. Testing showed decrease in execution time, more robust and accurate lane detection and tracking for every region of interest, workload balancing according to the area of expertise where CPUs are performing utility tasks and hardware accelerators mostly work on image processing.

Contents

Acknowledgements ... 7

Abbreviations ... 13

Introduction ... 17

1.1 Motivation ... 17

1.2 Problem Statement ... 19

1.3 Contributions ... 19

1.4 Evaluation of the performance ... 20

1.5 Thesis Outline ... 20

Background ... 21

2.1 Related Work ... 21

2.1.1 Lane Detection and Lane Tracking ... 21

2.2 FPGA, Altera's Cyclone® V, Stratix® V FPGA ... 25

2.2.1 FPGA ... 25

2.2.2 Benefits ... 26

2.2.3 Altera's Cyclone® V FPGA ... 26

2.2.4 Altera's Stratix® V FPGA ... 27

2.3 NVIDIA GeForce GTX 660 TI ... 28

2.4 Heterogeneous Platforms ... 29

2.5 Gaussian function smoothing ... 29

2.5.1 Image Convolution ... 30

2.6 Intel® Threading Building Blocks library ... 31

2.7 OpenCL ... 32

2.8 OpenCL – Runtime API ... 33

2.8.1 Create Buffer Objects ... 33

2.8.2 Read, Write Buffer Objects ... 34

2.8.3 Kernel Arguments and Queries ... 36

2.8.4 Kernel Execution on a Device ... 37

2.9 Particle Filter ... 38

Implementation ... 41

3.1 Overview of the Method ... 41

3.2 Details of the implementation ... 43

3.2.1 Pre-Processing ... 43

3.2.2 Elimination of unnecessary objects by Botsch 2015 ... 43

3.2.3 Elimination of unnecessary objects in current work 45
3.2.3.1 Using parallel execution on the host 45
3.2.3.2 Using OpenCL kernel 45
3.2.4 Grayscaling 47
3.2.5 Edge detection 48
3.2.6 Thresholding 49
3.2.7 Summary of the Pre-processing 50
3.3 Lane Detection 52
3.4 Lane Tracking 53
3.4.1 The Lane Tracking Algorithm 54
3.5 Redetection cases 58
3.6 Angle calculation 59
3.7 Further contributions 61
3.7.1 Multiple regions of interests 61
3.7.2 Adaptive regions of interest 62
3.8 Summary 63
Results 65
4.1 Chapter outline 65
4.2 Composition of the computation time 66
4.2.1 Initial Performance 66
4.2.2 Parallel processing by host 67
4.2.3 Parallel processing by kernel 68
4.3 Testing the algorithm on datasets 69
4.3.1 Lane Detection 70
4.3.2 Lane tracking 71
4.3.3 Threshold 73
4.3.4 Adaptive ROIs 74
4.3.5 Computation Speed 75
4.4 Known problems 78
4.5 Summary 79
Conclusion and Future Work 80
5.1 Conclusion 80
5.2 Future Work 82
Bibliography 84

Abbreviations

ADAS	Advanced Driver Assistance System
API	Application Programming Interface
CUDA	Compute Unified Device Architecture
DSP	Digital Signal Processor
LKWS	Lane-keeping and warning systems
LDAS	Lane Departure Warning System
DAS	Driver Assistance System
ECU	Electronic Control Unit
FLOPS	Floating Point Operations per Second
FPGA	Field Programmable Gate Array
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HHPC	Heterogeneous High Performance Computing
HPC	High Performance Computing
LDS	Lane Detection System
LDTS	Lane Detection and Tracking System
OpenCL	Open Computing Language
ROI	Region of Interest
PF	Particle Filter

List of Figures

2.1 From left to right: original image, morphological gradient of original image 16
2.2 Watershed of gradient image. Noise and inhomogeneity causes appearance of many catchment basins (not possible to detect the lanes) .. 16
2.3 Watershed of the gradient of the filtered image 17
2.4 Examples of likelihood detecting lines 17
2.5 Lane detection on a straight road 18
2.6 The road or lane boundaries can be extracted from the input image through a gradient Thresholding operation........ 18
2.7 The road region is a patch of shadow or sunlight........ 18
2.8 B-Snake based lane model. Left: using 3 control points. Right: using 4 control points 19
2.9 2D Gaussian curve........ 24
2.10 From left to right: the kernel (Gaussian), original image (matrix), and result values (matrix) 25
2.11 Lane Tracking........ 33
3.1 Structure of the Method........ 34
3.2 Image before and after preprocessing........ 35
3.3 Region of Interest........ 36
3.4 Distribution of the computation time in Botsch 2015 37
3.5 Input and output of the Grayscaling phase 40
3.6 Edge Detection 41
3.7 Before and after Edge detection........ 41
3.8 Thresholding........ 41
3.9 Before and after Thresholding is applied 42
3.10 Summary of preprocessing phase 43
3.11 The general flow of the algorithm........ 44
3.12 Lane Sampling 45
3.14 Phases of Lane Detection........ 46
3.15 The result of lane tracking phase are exact x and y coordinates of lane markings........ 47
3.16 Lane tracking is performed in three phases........ 47
3.17 Redetection cases 51
3.18 Best lines across all ROIs approximately form one line 52
3.20 Scheme of Connection detection between ROIs 52
3.21 Scheme of calculation of inclination angle 53

3.23 Lane Detection/tracking on corrupt frames 54
3.22 The process of processing ROIs 54
3.24 Adaptive ROIs 55
4.1 Distribution of workload and computation time in Botsch 2015 59
4.2 Distribution of the computation time in Botsch 2015 60
4.3 ROI selections on the host with parallel loops 61
4.4 Distribution of computation time with parallel loops 61
4.5 ROI selections on kernel 62
4.6 Distribution of computation time within Preprocessing 62
4.7 Lane Detection 64
4.8 Lane Tracking 65
4.9 Testing performed on TUM_DLR dataset with various threshold values. 67
4.10 Algorithm with adaptive ROIs 68
4.11 Performances on GPU 70
4.12 Performances on GPU of Botsch, 2015 70
4.13 Known issues 71

List of Tables

4.1 Testing settings.. 63
4.2 Distribution of computing time for frame with three non-adaptive ROIs. 68
4.3 Distribution of computation time for a frame with three adaptive ROIs.... 68

List of Algorithms

2.1 Algorithms of Particle filter.. 33
3.1 ROI selection on KERNEL.. 40
3.2 For performing Thresholding... 43
3.3 Preprocessing stage... 44
3.4 Particle Filter for Lane Tracking.. 48
3.5 Resampling algorithm.. 51

List of Formulas

3.1 Horizontal and vertical image convolution.. 42
3.2 Mathematical representation of Particle Filter.. 49

Chapter 1

Introduction

1.1 Motivation

Advanced Driver Assist Systems (ADAS) - research labs are currently developing sensor-based solutions to increase vehicle safety at lower speeds (when the driver is stuck in traffic), or at higher speeds (on a long highways). These systems, are known as Advanced Driver Assist Systems (ADAS). They combine stereo cameras, long- and short-range RADAR along with actuators, ECUs, and embedded software, to facilitate drivers to respond to changes in the environment quickly. Lane Detection and Tracking systems as well as many other solutions are part of ADAS.

There is a need in such systems because: according to the report of Federal police published in 2015 road accidents occurred two million times and is the main cause of deaths in Germany. Leading to 9,659 traffic deaths every year and most of these accidents are due to "human error"(75 %). This has a huge impact on wellbeing of society, family budget and lives of people since more than a million adult drivers or passengers are treated in hospitals due to card accident related injuries. According to the annual report car accident related expenses cost billions of Euro annually.

This pursues car manufacturers like BMW, to invest into research focused on vehicle safety. The ultimate goal is to start manufacturing “crashless” cars with build it Advanced Driver Assistant System which would provide comfort and security to drivers.

On the other hand, further research in this direction will allow taking automotive industry to the next level, where Germany will be able to hold a leading role in automotive industry. The vision of car manufacturers is to start producing semi-autonomous, fully autonomous cars.
This is important because Germany is known as one of world's top car exporter, major car manufacturers like Audi, Mercedes, Daimler, BMW, and Volkswagen are from Germany, the economy of the country is dependent heavily on export of automobiles therefore to be able to sustain leading positions it is absolutely vital to poses cutting edge technology.

The technology which would allow an average commuter spending at least 150 hours a year behind the wheel of a car to spend that time more efficiently, and invest it into something more meaningful, the least efficient use of that time would be to have a breakfast on the way to office, get enough rest on the way home, and work on report during a traffic jam.

All these poses more responsibility upon ADAS. Such systems are expected to be reliable and fast enough to be able to meet hard real time requirements. There is a huge work done in the field of Lane-keeping and warning systems, such systems exist in many variations showing very promising results. Most of them are based on image processing - processing streams of frames taken from a camera mounted on a vehicle and electronic control units (ECUs) - performing mainly image processing computations. However the recent trend is to replace ECUs with more efficient heterogeneous platforms combining advantages of hardware accelerators and conventional ECUs.

Porting of a legacy code into heterogeneous platforms is not an easy task and leads to bottlenecks and inefficiencies in the software, which was primarily designed for conventional ECUs and is being, used on heterogeneous hardware accelerators. LDTS system is not an exception and thus has to be "adjusted" or in most cases redeveloped to fit into new computing hardware and perform at its peak.

Existing algorithm done by Madduri 2014 and Botsch 2015 were not completely adapted to heterogeneous systems and therefore suffer from bottlenecks and other performance degradation factors. Target of this work is to continue adapting LDTS algorithm for the use on heterogeneous hardware accelerators, eliminate programming bottlenecks and introduce new features like angle calculation, view from quadcopter, multiple adaptable regions of interest and etc.

1.2 Problem Statement

This thesis serves as an extension to a previous work done by Madduri, 2014 and Botsch 2015. In this thesis further improvements on Lane Detection and Tracking algorithm were done, taking into consideration **Future Work** section proposed by Botsch 2015 and new requirements from BMW research lab. Though many important milestones have been achieved by previous developers, the discussion among lab's representatives and members of Chair of Robotics and Embedded Systems revealed new areas requiring further research and experiment.

In addition, due to time constraints or lack of expertise work by Botsch 2015 did not provide much room for flexibility (ex. number of regions of interest or their size) and did not replace all major segments of code, programming bottlenecks degrading performance by OpenCL kernels.
Hence, it was decided to invest more effort and time into the Botsch 2015 work and take it to the next level, where new algorithm would tackle challenges offered by above mentioned organizations and would deliver best possible execution time.

The algorithm is based on computer vision, does not require any special settings for a camera, except that it should be installed on top of the vehicle and provides a long range detection and tracking of the road lane markings.

1.3 Contributions

The major contributions of the current thesis are:

1. Elimination of programming bottlenecks, segments of code hampering execution time.
2. Enabling support for multiple and independent regions of interests for each frame.
3. Calculation of angle of a lane detected on multiple regions of interest, thus enabling DAS to deal with road bending and sharp turns in advance.
4. Develop an algorithm supporting adaptable regions of interest, thus significantly reducing computation effort.

1.4 Evaluation of the performance

The following hardware was used for developing and testing the algorithm:

- GPU: NVidia GeForce GTX 660 TI
- FPGA: Altera Cyclone V
- WORKSTATION: RAM 32 GB, Intel Core i8
- QUADCOPTER Parrot Bebop Drone, 1920x1080p, 30 Frames per/s

Video recorded from quad copter and independent datasets were used to test the performance of the algorithm thoroughly.

1.5 Thesis Outline

Chapter two provides a theoretical background on technology used in this thesis to improve lane detection and tracking algorithm. It also provides an explanation of why Particle Filter has been used, why Lane Detection and Tracking algorithm works better with OpenCL kernels on hardware accelerators, what is the role of Gaussian function in this algorithm, why there is a trend towards heterogeneous hardware platforms and etc.

Chapter three introduces changes done to algorithm written by Botsch 2015, explains the reasoning behind new lane detection and lane tracking algorithm. It explains how placement of multiple regions of interest on a single frame was achieved. Gives insight into road angle calculation, elimination of programming bottlenecks, support of adaptive regions of interest

Chapter four shows results of comprehensive testing done on different datasets and pre-recorded videos using *Quadcopter Parrot Bebop Drone*, draws results and compares them to the results obtained in previous work.

And finally chapter five provides a summary of the project and concludes with suggestions on what could be the next step for Lane Detection and Tracking algorithm and Driver Assistant Systems in general.

Chapter 2

Background

2.1 Related Work

2.1.1 Lane Detection and Lane Tracking

In order to keep the position of the vehicle within boundaries of road lanes, it is necessary to measure location of the vehicle along the lines. Many kinds of methods have been proposed and tested for this purpose, such as:

- on-board vision systems for detection of the painted lane markings;
- continuous magnetic wires integrated into the center of the lane mark;
- measurement of the distance to sidewalls using radar or ultrasonic waves;
- detection of reflective markers by means of laser technologies and etc.;

Most of above mentioned methods require necessary highway infrastructure to be built, except the on-board vision systems which are more complicated in development but have an advantage in capability of autonomous operation.

Many different vision-based lane detection algorithms developed to date depend on different road models (2D or 3D, straight or curve) and different techniques (Hough transform, template matching, neural networks, machine learning and etc.).

In most cases these vision-based lane detection systems follow next steps:

1. Capturing the frame from camera,
2. Separation of the image into necessary amount of region of interests,
3. Lane detection process,
4. Defining the lane markings.

The process for detection of the lane has a significant number of solutions in several works published in the literature.

The solution shown in papers [2] [3] describes the method which uses principal of morphological filtering. In this techniques in order to locate the lane edges in the intensity gradient magnitude image the "watershed" transformation is being used. The idea of the "watershed" transformation can be described as a landscape sunk in a lake, with holes located in local minima. Starting from this points (local minima), catchment basins are filled with water. After that the dams are being built at the boundary points where water comes from different catchment areas. This process continues until the level of water will reach the highest peak in the landscape. As a result obtained landscape is separated into regions by dams which are called watershed lines or "watersheds". The strength of this technique is the fact that there is no need in any thresholding of the gradient magnitudes but at the same time it has a significant lack in not establishing any global constraints on the shape of the lane edge.

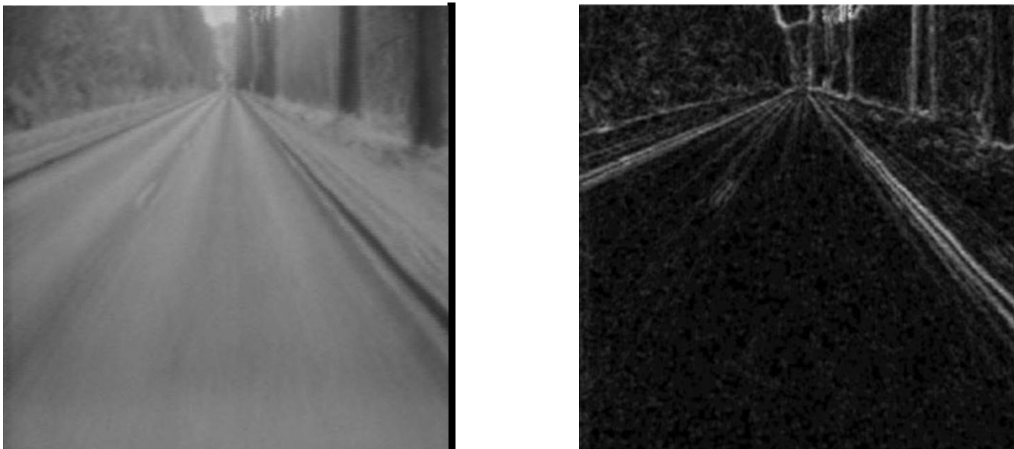


Figure 2.1: From left to right: original image, morphological gradient of original image

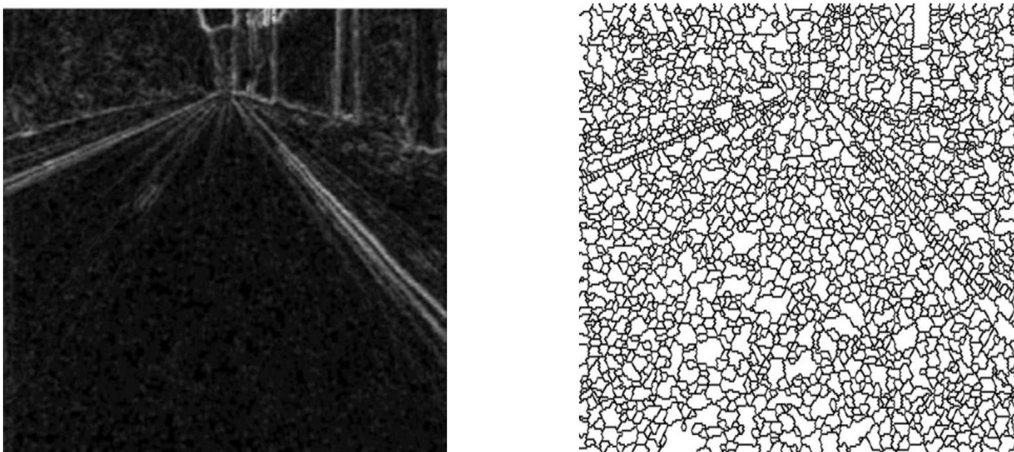


Figure 2.2: Watershed of gradient image. Noise and inhomogeneity causes appearance of many catchment basins (not possible to detect the lanes).

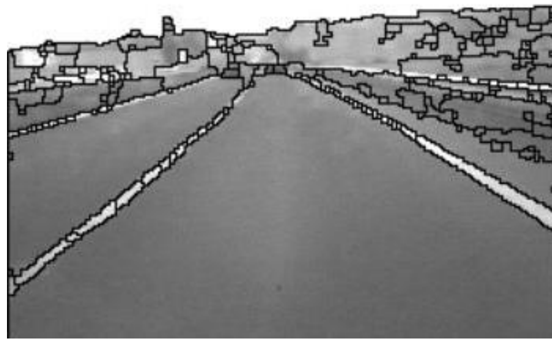


Figure 2.3: Watershed of the gradient of the filtered image

The approaches in [4] [5], proposes that parabolic curve could be the way in which the lane borders can be described on smooth ground. Even though this method can approximate conventional road schemes, it will not be able to recreate such an example of road schemes as a T -shaped intersection. Based on this model and on the improvement of the likelihood function deformable template method was proposed.
However this technique does not provide high accuracy and global optimum without huge computational power.

Figure 2.4 shows examples of deformable template method based on likelihood (LOIS) lane detection function under a variety of road and environmental conditions.

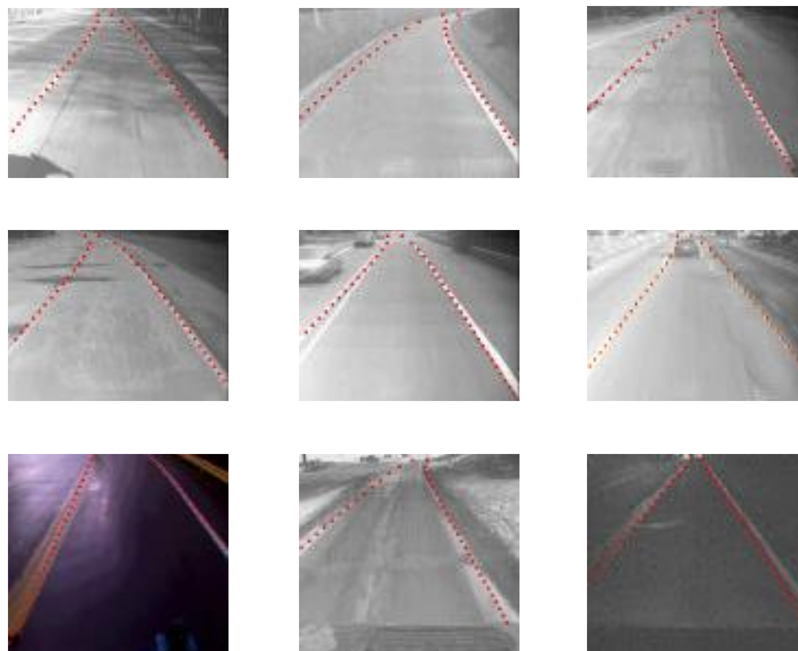


Figure 2.4: Examples of likelihood detecting lines

Recognition algorithm based on road edges is considered in articles [6][7][8][9]. In spite of the conditions associated with the shadows on the road this method deals with the problem quite well in well painted road marks, but for roads with not painted road markings (Figure 2.7), where side of the road will have to be determined by the boundaries of the road, this algorithm is not suitable.

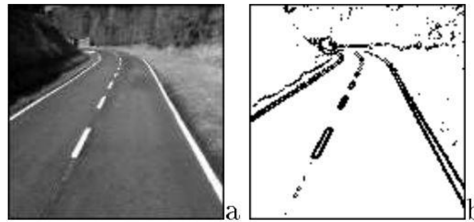


Figure 2.5: Lane detection on a straight road: (a) input image; (b) image obtained by thresholding the gradient image.

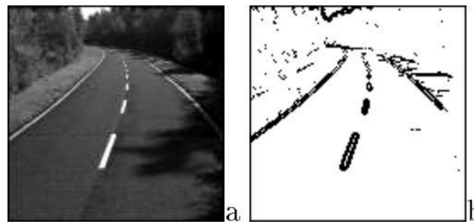


Figure 2.6: The road or lane boundaries can be extracted from the input image through a gradient thresholding operation

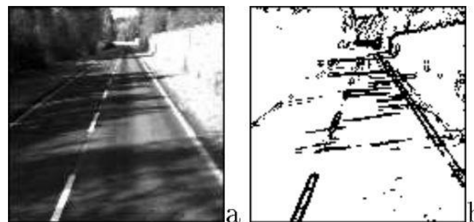


Figure 2.7: The road region is a patch of shadow or sunlight

The method of combining the Hough transform and Line-Snake model is considered in the article [11]. The method is based on the separation of the image into several regions in the vertical direction. In order to get the initial position estimation of the lane boundaries on the surface of the road the Hough transformation should be applied for each region. Further, the model of Line-Snake is being used, the purpose of which was improvement of the initial approximation to a better configuration of the lane boundaries. However, this technique has two major problems. In the first case the mark-up line is with breaks, it is likely that it will not continue to the bottom of the picture. In the second case, the contrast of the line edges at the distance close to the bottom of the image may not be sufficient for correct detection.

The figure 2.8 represents the description of the road shapes using different number of control point in the B-Snake based lane model

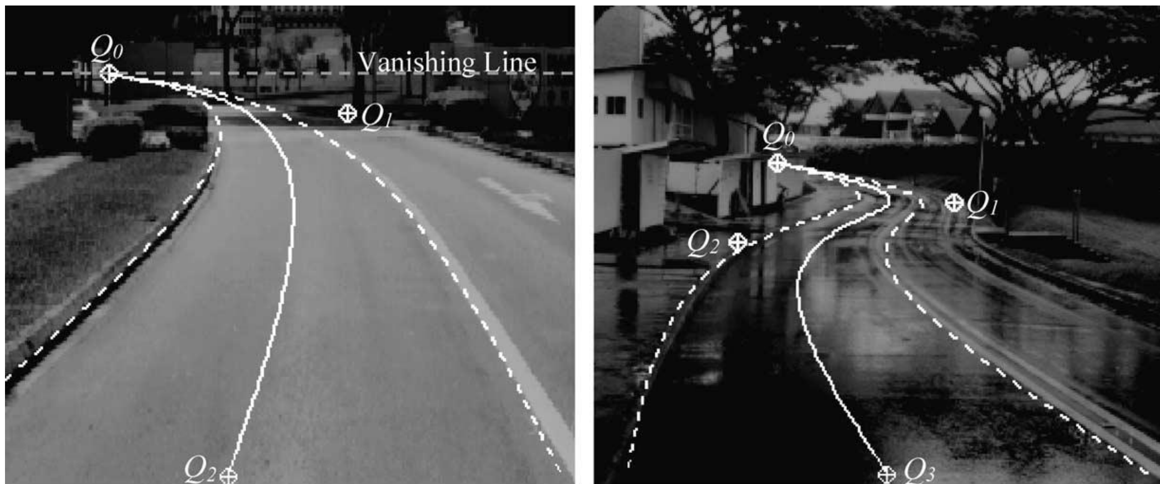


Figure 2.8: B-Snake based lane model. Left: using 3 control points. Right: using 4 control points.

Lane boundary recognition method based on artificial vision is presented in papers [12] [13]. This method is used for country roads. In order to detect the difference between road and non-road area the method uses statistical criteria like energy, contrast and homogeneity. However, while applying the same road model used in [4] [5], the same problems are being faced.

2.2 FPGA, Altera's Cyclone® V, Stratix® V FPGA

2.2.1 FPGA

In older times only engineers with deep knowledge in the digital hardware design were able to cope with the application of FPGA technology. New technologies make it possible to convert graphical diagrams and C code into digital hardware circuitry, thus making FPGA a reprogrammable silicon chip. Another distinctive feature of FPGAs is that they are not limited by the availability of the number of processing cores. FPGAs are parallel in nature, which distinguishes it from other processors. This means that the various tasks being processed are not performed using the same resources. Each task is processed independently of each other, individually assigned to certain part of the chip and can be operated without any interference from other logic blocks. Ultimately, regardless of adding more tasks being processed, the performance of one part will not have any influence on others [15].

2.2.2 Benefits

1. **Performance**— FPGA has an advantage of parallel execution on hardware level that allows it to break the rule of sequential execution and handle more processes per cycle.
2. **Reliability**—Systems based on processors mainly resort to the method of resource sharing between different processes. Such systems have layers between parts of the system. In order to control the hardware resources driver layer is being used. Only one task can be executed at one time for each processor core, and therefore the system based on processors have the risk of collision with the offloaded tasks and loading another. FPGA does not use the OS so the risks with reliability is minimized due to parallel execution of tasks and a separate dedicated hardware for each task
3. **Long-term maintenance**—FPGA chips do not require much time and manufacturing costs, because FPGAs are fully upgradable, and it is possible to change the configuration at any time [15], thus making them suitable for on board ADA systems.

However, as with all systems, in case of FPGA, there are a number of shortcomings that need to be noted:

1. The cost is much higher than custom silicon
2. Higher power consumption in comparison with ASIC.
3. Compared with general-purpose processor, FPGAs require longer configuration and compilation time

2.2.3 Altera's Cyclone® V FPGA

Intelligent video analysis in real time is an integral part of systems like, advanced driver assistant systems, industrial computer vision, robot motion planning and so on. These systems require complex algorithms for executing motion detection, object recognition, image processing tasks. Using Altera SoCs (**S**ystem **o**n a **C**hip), developers get a great tool, where only one chip contains power of the FPGA and dual-core ARM® Cortex®-A9 HPS (Hard Processor System). Developers are able to optimize complex algorithms by transferring intensive computing functions of HPS in FPGA, thus increasing system performance [14]

The Cyclone® V devices meet the requirements of reducing energy costs, time to market and a growing range of applications sensitive to costs. With the presence of built-in memory controllers and transceivers, the Cyclone V development kits is ideal for use in areas such as industry, wireless and wireline, military, automotive control systems and driver assistant systems.

Main Advantages of Cyclone V Devices:

Advantage	Feature
Reduced energy consumption	• Built on TSMC's(Taiwan Semiconductor Manufacturing Company)28 nm low-power (28LP) process technology and includes an abundance of hard intellectual property (IP) blocks • In comparison with previous generation, in this one power consumption decreased up to 40%
Improved logic integration and differentiation capabilities	• Adaptive Logic Module (ALM): 8-input • Embedded memory: ≈ 13.59 megabits (Mb) • Variable-precision digital signal processing (DSP) blocks
Increased bandwidth capacity	• 3.125 gigabits per second (Gbps) and 6.144 Gbps transceivers • Hard memory controllers
Hard processor system (HPS) with integrated ARM® Cortex™-A9 MPCore processor	• Integration in a single Cyclone V SoC (system-on-a-chip) of an FPGA, dual-core ARM Cortex-A9 MPCore processor and hard IP • Supports over 128 Gbps peak bandwidth with integrated data coherency between the processor and the FPGA fabric
Lower system expenses	• For operation needs only two core voltages • Low-cost wirebond packaging is supported • Includes innovative features such as Configuration via Protocol (CvP) and partial reconfiguration[16]

2.2.4 Altera's Stratix® V FPGA

One of the main features of the Altera's 28-nm Stratix® V FPGA is enhanced core architecture, built in transceivers with the working speed up to 28.05 gigabits per second and integrated hard intellectual property (IP) blocks in the form of unique array.

This kind of improvements gives a new class of Stratix V FPGA, which is optimized for application targeted devices:

- Bandwidth-centric applications and protocols, including PCI Express® (PCIe®) Gen3
- 40G/100G data-intensive applications
- Application for high-performance and high-precision digital signal processing (DSP).

Stratix V devices has a four variants (GT, GX, GS, and E), and each of them are targeted for different (specific) set of applications. In general Stratix V FPGA is being used for higher quality production, and for low risk, low cost production mainly HardCopy® V ASICs is used.

Like in all Stratix V family variants there are a rich set of high-performance building blocks which has a redesigned ALM (adaptive logic module), embedded memory blocks for 20 Kbit, DSP (Digital Signal Processing) blocks, fractional PLLs (Phase-Locked Loops). Altera's architecture based on multi-track routing concept interconnects all above mentioned building blocks. Also one of the main features of Stratix V devices is the new built in HardCopy Block, which achieves Altera's unique HardCopy ASIC abilities as it has customizable hard IP bock.

2.3 NVIDIA GeForce GTX 660 TI

NVIDIA GeForce GTX 660 TI has dozens of cores and these cores are different from cores in CPU. NVIDIA GeForce GTX 660 TI cores are designed for intensive graphics related computations and can do image processing work faster than CPU. Also, it is possible to apply parallel execution due to the number of GPUs in current graphic cards.
GPUs are able to have much higher number of transistors compared to CPUs and do not have to deal with cache and control logic, but are mainly focused on graphics processing tasks [18] [17]. Some tasks in Lane Detection and Tracking algorithm (example: selection of ROI, resampling, thresholding) are executed on NVIDIA GeForce GTX 660 TI because pixels in the original image are independent and computations require no or very little synchronization in between, thus can be proceed in parallel [19]. This is the reason behind significant increase in computational speed after delegating image processing tasks to hardware accelerators (more about it in Chapter 3).

The drawback of GPUs is that they require special programming models. For instance, NVIDIA GeForce GTX 660 TI cards used in this thesis should be programmed either with CUDA or OpenCL.

2.4 Heterogeneous Platforms

Heterogeneous Platforms - are the workstations with a mix of different types of cores or processors (often CPUs and GPUs) where GPUs and FPGAs are normally used as additional processors to a CPU. The reason for combining different types of processors under one platform is that CPUs are best suited for sequential tasks [37] or OS related peripheral tasks and computationally intensive graphics processing operations are for hardware accelerators.

An average CPU has four, eight, sixteen cores, has about a billion transistors and can achieve about 0.5 TFlop for floating point calculations. An average GPU has 64 cores, is 64x-threaded, and has on average twice as many transistors. GPU can achieve 6 TFlop — 120 times of CPUs processing speed.

By combining both under one platform it is possible to delegate computationally intensive tasks to NVIDIA GeForce GTX 660 TI, ALTERA Stratix V or ALTERA Cyclone V SOC, while CPU can run the OS and handle other peripheral tasks.

Heterogeneous systems are typically used in computer vision, robot motion planning or simulation tasks. For example, Lane Detection and Tracking algorithm developed at TU Munich is based on heterogeneous platform where GPU and FPGAs took a role of hardware accelerators connected to a host CPU system. Host system runs the main application with all the peripheral tasks and delegates' graphics processing tasks to hardware accelerators. The probable drawback of this architecture is in certain delay for host to kernel communication but execution time profiling revealed that overheads are insignificant, unless performed too often.

2.5 Gaussian function smoothing

Image like as in the case of one-dimensional signal can be obtained with some noise and for eliminating it before the main processing of the image starts, pre-processing filters are normally applied.

Normally, the noise is regarded as an arbitrary combination of the colour and brightness information, which should not be present in the original image. The process of the emergence of noise can be due to sensor and chips errors in digital

cameras. The idea of using low-pass filter kernels is in smoothing the image. The high-frequency part of the image such as edges and noise are removed by the low-pass filter, and some image processing techniques are applied to prevent blurred edges.

The Gaussian function is based on the usage of averaging filter except for the fact that instead of using the single weight for each pixel, a two-dimensional Gauss function applied for the kernel that provides the highest weight to the pixel located in the centre.

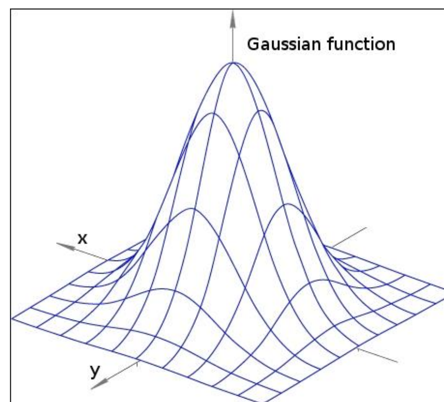


Figure 2.9 2D Gaussian curve

One of the forms of mathematical convolution is 2D Kernel Convolution. The resulting image is calculated by iterating over each pixel of the original image and applying kernel to it, which ultimately gives us a new pixel for each operation. For example, if the kernel operator is 3 x 3 matrixes, the final pixel will be an average value of 9 adjacent pixels for each pixel of the input image, resulting in an averaged output image.

2.5.1 Image Convolution

The main part of the majority of filtering operations is based on image convolution. The basic idea of image convolution is the application of image transformation technics based on neighbouring pixels to each pixel of the original image. For this transformation matrix which is simple 2D matrix is used, and for the sake of simplicity, this matrix is called a Kernel. The new value of the resulting image is being calculated as the sum of the products for each pixel of the original image. To compute these products, the multiplication of the kernels with the appropriate pixel of the image should be done, and also the central element of the kernel must be multiplied with the actual image pixel.

Practical implementation of convolution can be shown on most popular filter which uses the Gaussian kernel. The practical application of the Gaussian filter (Gaussian blur) is very wide and can be used for image smoothing, noise removing and edge detection. The edge detection algorithm in most cases is very sensitive to noise. Therefore Gaussian filter is applied for eliminating unnecessary noise before the actual edge detection is performed.

The next figure shows the convolution example using the Gaussian kernel:

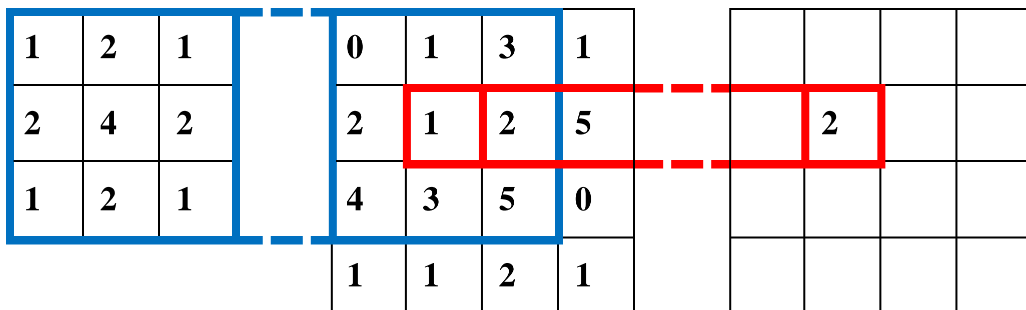


Figure 2.10: From left to right: the kernel (Gaussian), original image (matrix), and result values (matrix).

For the computation of the value in the position (2, 2) of the resulting matrix, the following process is applied:

(1*0 + 2*1 + 1*3 + 2*2 + 4*1 + 2*2 + 1*4 + 2*3 + 1*5) / 16 = 2

2.6 Intel® Threading Building Blocks library

For experimentation purposes it was decided to try to re write sequential code using parallel programming techniques offered by Intel threading building blocks library. If computational results are satisfactory, there will be no further need in developing OpenCL kernels. Thus segments of code occupying bulk of computation time due to sequential execution on host could still remain on host but instead would be executed in parallel. This would help to save time, as writing kernel code, handling memory issues, integrating kernels into the main application and recompiling executables is not an easy task.

Intel threading building blocks library enables developers to write parallel applications in C++. The well-known advantage of the Intel TBB library is that it

makes parallel performance and scalability accessible to developers, especially for those, writing loop based applications (ROI selection consumes bulk of computational resources and is performed by two nested loops iterating through rows and columns of the original image and processing image pixels one by one in a sequential manner) [35].

In TBB it is possible to select ROI by wrapping the serial loops into one parallel_for. parallel_for divides index space into sections based on the grain size, which is passed as argument. TBB creates and schedules threads to run above mentioned sections of work on its own, it promises to improve efficiency by assigning available worker threads to work items, by making sure that no thread stays idle.

No thread stays idle because TBB implements work stealing to divide workload across available cores. This approach helps to increase core utilization and scaling. In TBB work stealing model, the workload is evenly divided among the available cores. If one core completes its work while other cores still have big amount of work in their queue, it reassigns some of the work from busy cores to idle ones [36].

2.7 OpenCL

OpenCL - is a framework for programming on heterogeneous high performance devices (ex. GPGPU, FPGA connected to CPU) which allows the developer to write C++ functions, called kernels. The framework provides a high level of abstraction to write low level hardware instructions.
Hardware accelerator consists of compute units and processing elements. Compute unit consists of compute kernels written in OpenCL C. Kernels contain sequence of instructions, which are called work item. Work group consists of several work items, which are executed concurrently.
OpenCL kernel is alternative to Intel® Threading Building Blocks library, with one major difference: it runs on hardware accelerators. Benefits of re writing some segments of code in Open CL kernels is that it will allow to process each pixel independently and in parallel to others on GPGPU/FPGAs.

Kernels obtain information from main application via the following memory regions:

- Global memory - all work items in all work groups have enough privileges to write to and read from this memory region

- Constant memory - a fraction of global memory, is only accessible by a host system and is not volatile.

- Work Group's local memory - work items belonging to work group are only able to access this part of memory

- Work item's private memory - accessible by work item only

Thread block on a current GPU contains up to 1024 threads. One kernel can be executed by several thread blocks therefore the total number of threads working on a single kernel (such as. ROI selection, Line sampling) is equal to number of threads in a single block multiplied by number of blocks. Thread blocks are not dependent on each other, can be executed in any order on any of the available cores thus enabling scalability tied to number of available cores.

2.8 OpenCL – Runtime API

2.8.1 Create Buffer Objects

The memory objects located in the main memory of the host or the global memory installed at the accelerator require careful treatment in OpenCL. One of the reasons of such condition is the slowness of these memories. Another reason is that the constant copying between these two memories takes a decent amount of time [23].

In programs written using OpenCL the buffer objects are applied to represent generic data. OpenCL provides the ability to transfer data without converting to OpenCL compatible device using buffer objects and then manipulate data using the familiar properties of the C similar languages. This approach eliminates the need to convert the data to a specific hardware format.

Since the data transfer consumes some time, the best option would be to minimize the reading and writing sessions as much as possible. It is possible reduce the amount of data traffic needed for processing data using the method of packaging of all host data in a buffer object that may remain on the device [24].

cl_mem **clCreateBuffer** (cl_context context,
cl_mem_flags flags,
size_t **size**,
void ***host_ptr**,
cl_int ***errcode_ret**)
flags: CL_MEM_READ_WRITE,
CL_MEM_{WRITE, READ}_ONLY,
CL_MEM_HOST_NO_ACCESS,
CL_MEM_HOST_{READ, WRITE}_ONLY,
CL_MEM_{USE, ALLOC, COPY}_HOST_PTR [25].

The function **clCreateBuffer** is based on the creation of OpenCL-specific object that is passed as an argument to the kernel.

The parameters can be described as:
size
The buffer memory object has to be allocated using size in bytes.
host_ptr
Is a pointer to the buffer data which is allocated by the application.
errcode_ret
Returns an appropriate error code. If it is NULL, no error code is returned.

2.8.2 Read, Write Buffer Objects

With the event mechanism in OpenCL it is extremely easy to manage different parts of algorithm. For example, memory objects can be transferred from the host memory to the memory of the devices and back using data transfer APIs **clEnqueueReadBuffer** and **clEnqueueWriteBuffer**.

With the application of API **clEnqueueWriteBuffer** it is possible to write to the device memory immediately before the launch of Kernel. In order to get back the buffer data from the device memory to the host upon completion of the processing of the kernel **clEnqueueReadBuffer** function is used [27].

cl_int **clEnqueueWriteBuffer** (
cl_command_queue **command_queue**,
cl_mem **buffer**,
cl_bool **blocking_write**,
size_t offset,
size_t size,
const void ***ptr**,
cl_uint **num_events_in_wait_list**,
const cl_event ***event_wait_list**,
cl_event ***event**) [28].

The function **clEnqueueReadBuffer** reads data from the device to the host memory:

cl_int **clEnqueueReadBuffer** (
cl_command_queue **command_queue**,
cl_mem **buffer**,
cl_bool **blocking_read**,
size_t offset,
size_t size,
void ***ptr**,
cl_uint **num_events_in_wait_list**,
const cl_event ***event_wait_list**,
cl_event ***event**) [29].

The parameters can be described as:

command_queue
Refers to the command-queue in which the read (write) command will be queued.

buffer
Refers to a current buffer object.

blocking_read
Indicates if the read operations are blocking or non-blocking. If operation **blocking_read** is true i.e. the read command is blocking, **clEnqueueReadBuffer** does not return until the buffer data has been read and copied into memory pointed to by **ptr**.
If operation **blocking_read** is false i.e. the read command is non-blocking, **clEnqueueReadBuffer** queues a non-blocking read command and returns. The contents of the buffer that **ptr** points to cannot be used until the read command has completed.

blocking_write

Indicates if the write operations are blocking or nonblocking.

If operation **blocking_write** is true, the OpenCL implementation copies the data referred to by **ptr** and enqueues the write operation in the command-queue. The memory pointed to by **ptr** can be reused by the application after the **clEnqueueWriteBuffer** call returns.

If operation **blocking_write** is false, the OpenCL implementation will use **ptr** to perform a nonblocking write. As the write is non-blocking the implementation can return immediately. The memory pointed to by **ptr** cannot be reused by the application after the call returns.

offset

The offset in the buffer object to read/write from (in bytes).

cb

The size of data being read/written (in bytes).

ptr

The pointer to buffer in host memory where data is to be read into.

event_wait_list , num_events_in_wait_list

event_wait_list and **num_events_in_wait_list** specify events that need to complete before this particular command can be executed.

event

Returns an event object that identifies this particular read command and can be used to query or queue a wait for this particular command to complete.

2.8.3 Kernel Arguments and Queries

It is not possible to invoke a kernel with a certain list of arguments, in contrast to a function call in the C++ programs. In order to start the kernel the scheduling through the initialization function of the queue should be applied. Each argument of the kernel must be specified separately with **clSetKernelArg ()** function (the syntax of C++ language allows it). It should also be noted that kernel arguments are persistent.

The inputs for this function are the object of the kernel, the argument number with an index, size of the argument and the pointer to the argument. Then, in order to make a correct extraction of the data according to the type, the information on the type should be applied from the list of kernel parameters [30].Arguments of the Kernel must be set prior to execution using the function **clSetKernelArg** [31] as follows:

cl_int **clSetKernelArg** (
cl_kernel **kernel**,
cl_uint **arg_index**,
size_t **arg_size**,
const void ***arg_value**) [32].

The parameters can be described as:

kernel
A current kernel object.

arg_index
The argument index. Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to n - 1, where n is the total number of arguments declared by a kernel.

arg_value
A pointer to data that should be used as the argument value for argument specified by arg_index.

arg_size
Specifies the size of the argument value [32].

2.8.4 Kernel Execution on a Device

Kernel is started with **clEnqueueNDRangeKernel()** function. The command-queue will be triggered successfully if the target device is already set. Object of the kernel in the main class identifies the executable code. In order to create a work item four fields are being used. The parameter which defines the number of dimensions on the basis of which work item is being created is called - **work_dim**. **global_work_size** describes the number of work items for each dimension in **NDRange** and **local_work_size** specifies the number of work items for each dimension of the working groups. Another parameter named **global_work_offset** can be applied to ensure the balance so that global identifiers (ID) of the working items do not start from zero [30].

cl_int **clEnqueueNDRangeKernel** (
cl_command_queue **command_queue**,
cl_kernel **kernel**,
cl_uint **work_dim**,
const size_t ***global_work_offset**,
const size_t ***global_work_size**,
const size_t ***local_work_size**,

cl_uint **num_events_in_wait_list**,
const cl_event ***event_wait_list**,
cl_event ***event**) [34].

2.9 Particle Filter

Particle filter [20] is a numerical approximation to the nonlinear Bayesian filtering problem and it's used during Lane tracking phase. It helps to decrease overall computational time of the lane detection algorithm by using the previously calculated information such as best_lines and good_lines for predicting the measurements for the next frame, thus to avoid allocating hundreds of sampling lines and weighting them again.

Particles in the particle set are the samples of posterior distribution and are represented by:

$$\mathrm{X_t} := \mathrm{x}_\mathrm{t}^{[1]}, \mathrm{x}_\mathrm{t}^{[2]}, \ldots\ldots, \mathrm{x}_\mathrm{t}^{[M]} \ ;$$

$$1 <= m <= M$$

Each particle $x_t^{[m]}$ represents a possible true state at time t. M - Represents the number of particles and $\mathrm{X_t}$ is a particle set. The trick of PF is to approximate belief $bel\ (X_t)$ - measurements obtained during lane detection phase, by user defined number M of particles. Bigger the M is, the more likely it is to track the lane accurately. Similar to other filters from the family of Bayes filter algorithms, PF obtains the right measurements for current frame $bel\ (X_t)$ recursively using the measurements obtained one time step earlier $\mathrm{bel}(x_{t-1})$. Belief is nothing but a set of particles, thus PF performs lane tracking by obtaining the set of particles X_t recursively from the previous set at an earlier time tx_{t-1}. In this algorithm the previous set at time $t\ x_{t-1}$ can be obtained by an empirical set of particles, such as **good lines** and b**est lines,** kept from a previous frame.

2.1 Algorithm Particle_filter (X_{t-1}, u_t, z_t)

1: $\bar{X}_t = X_t = \emptyset$
2: **for** $m = 1$ to M **do**
3: Sample $x_t^{[m]} \sim p(x_t | u_t, x_{t-1}^{[m]})$
4: $\omega_t^{[m]} = p(z_t | x_t^{[m]})$
5: $\bar{X}_t = \bar{X}_t + \langle \omega_t^{[m]}, \omega_t^{[m]} \rangle$
6: **Endfor**

7: **for** $m = 1$ to M **do**
8: draw i with probability $\infty \omega_t^{[i]}$
9: add $x_t^{[i]}$ to X_t
10: **end for**
11: **retrun** X_t

x_{t-1} - Particle set, u_t - the recent control, z_t - are the latest measurements [20]

In the beginning the algorithm constructs a temporary particle set X, which is similar to the belief $bel\ (X_t)$ by processing each particle $x_{t-1}^{[m]}$ from the particle set.

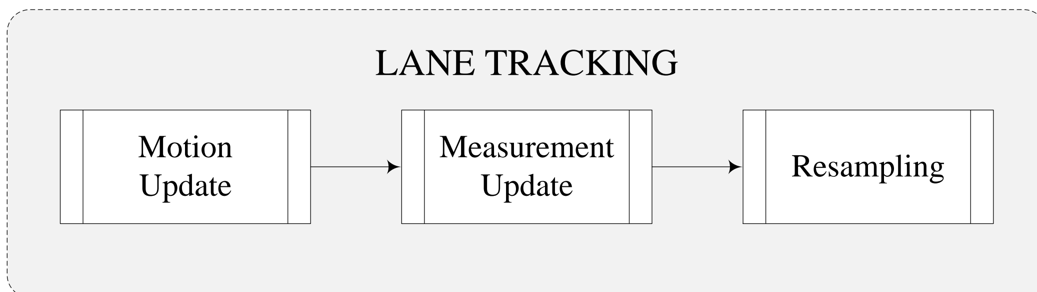


Figure 2.11 Lane Tracking

1. Line 3 assigns a probable state $x_t^{[m]}$ for time t based on the particle $x_{t-1}^{[m]}$ of the previous frame. The resulting sample is labeled by m, to show that it is obtained from the m-th particle in x_{t-1} - of the previous measurement. The set of particles obtained in step 3 is the filter's representation of bel(x_t) and corresponds to **motion update** on Figure 2.11
2. Line 4 calculates weight of a particle. Weights of a particle are used to include the latest measurements into the particle set. The set of weighted particles represents posterior bel (x_t), and corresponds to **measurement**

update step of Figure 2.11. The weights will vary based on how likely the particles represent the true lane.

3. Lines 8 through 11 implement resampling or importance resampling by drawing each particle by its importance weight. Resampling transforms a particle set of M particles into another particle set. By incorporating the importance weights into the resampling process, the distribution of the particles changes: whereas before the resampling step, they were distribution according to bel (x_t), after the resampling they are distributed (approximately) according to the posterior bel (x_t). Resampling step ensures survival of the fittest.

Chapter 3

Implementation

3.1 Overview of the Method

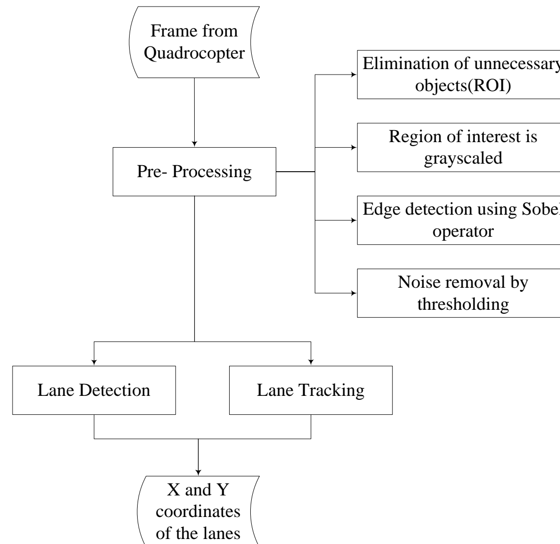


Figure 3.1: Structure of the Method

As shown on Figure 3.1 algorithm works with stream of frames coming from a camera installed on top of a vehicle or quadcopter hovering over the car. It consists of three main stages: preprocessing, lane detection and lane tracking.

Preprocessing stage comprises of several phases. In *Elimination of unnecessary objects phase* color image is cleared of objects of least interest, such as sky, trees, buildings and etc. by selecting the region of image where lane markings are most likely to be found.

In the next step selected region is Grayscaled, mainly for practicality reasons. After obtaining black and white image edge detection and Thresholding operations are performed. The final output of preprocessing stage is shown in Figure 3.2b.

Execution time profiling revealed that programming bottlenecks within preprocessing phase occupied bulk of execution time therefore this phase has been completely re-written two times, first using parallel programming techniques in C++, to run on host machine, which later proved to be not sufficient, and eventually in OpenCL kernel, which allowed executing this stage completely in parallel on hardware accelerators. Thus sections of code hampering execution time of this stage were completely eliminated.

a) Original frame b) Preprocessed frame

Figure 3.2: Image before and after preprocessing

As can be seen from Figure 3.1, following the preprocessing phase, depending on availability of previous weighing, either lane detection or lane tracking is performed. In case of lane detection - a random sampling is executed to sample probable road lane markings, lines are weighted according to their distance to the lane and line with the highest weight is selected to represent the real lane.

In case of lane tracking - the weighing from a previous frame along with a Particle Filter are used to detect the road lane markings in the current frame. Lanes are not evaluated again but are tracked, thus the name Lane Tracking.

Lane detection/ tracking are performed by OpenCL kernels, in parallel [because lines are completely independent] and computations are delivered by hardware accelerators.

Subsequent sections will provide a detailed explanation of above mentioned stages, along with illustrations and achieved results.

3.2 Details of the implementation

3.2.1 Pre-Processing

Objects of least interest such as sky, trees and buildings are discarded by selecting regions where road lane markings are most likely to be located. As can be seen on Figure 3.3 this area is surrounded by a green rectangle and called a region of interest. Further stages of algorithm work with ROIs only and discard everything outside of the green rectangle.

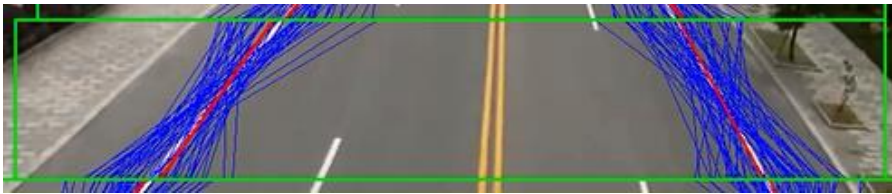
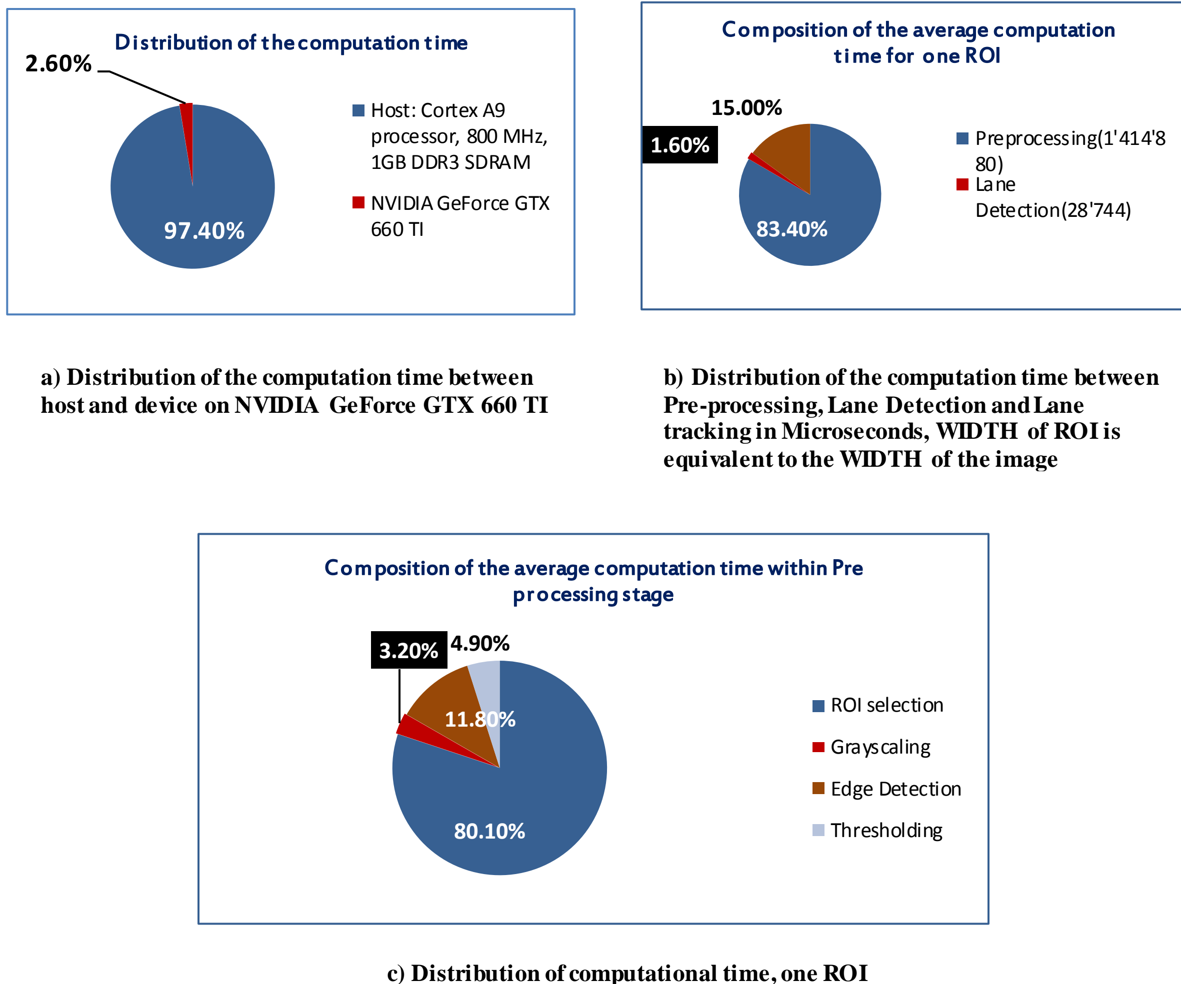


Figure 3.3: Region of Interest

3.2.2 Elimination of unnecessary objects by Botsch 2015

In Botsch, 2015 ROI was selected in a sequential manner and was performed by host machine. Image was first processed by rows and then by columns. In each iteration of the loop calculations on color channels were performed, bit shifting was done and the values of three channels were combined. This worked pretty well for a single ROI but proved to be unacceptable for multiple ROIs.
As can be seen on the Figure 3.4b, with one ROI where *ROI.WIDTH* = *original_image.width,* pre-processing phase alone occupied more than 83% of computation time and the remaining phases like Lane Detection and Lane Tracking took only 1.6% and 15% respectively. The reason for preprocessing stage consuming considerable computational time is that ROI selection was done in sequential manner on host.

Execution time profiling revealed that within preprocessing stage 80% of the computational time was occupied by selection of ROI, and the rest of the operations like Grayscaling, Edge detection and Thresholding took only 20% of the time. Exact distribution of computational time is shown on Figure 3.4.



a) Distribution of the computation time between host and device on NVIDIA GeForce GTX 660 TI

b) Distribution of the computation time between Pre-processing, Lane Detection and Lane tracking in Microseconds, WIDTH of ROI is equivalent to the WIDTH of the image

c) Distribution of computational time, one ROI

Figure 3.4 Distribution of the computation time in Botsch 2015

Besides that, as shown on Figure 3.4a distribution of the workload between host and kernel is 97.40% and 2.60% respectively. The distribution is strongly uneven, host performs bulk of image processing task though hardware accelerator would have done it more efficiently. One of the factors contributing heavily to such uneven workload distribution is the same - ROI selection was performed by host, in a sequential manner.

Above mentioned issues had to be eliminated, otherwise allocation of multiple ROIs for each frame would increase computation time many folds, and the whole algorithm would not be able to meet hard real time requirements. The following sections will explain the work done and resolved issues.

3.2.3 Elimination of unnecessary objects in current work

Elimination of unnecessary objects phase has been completely re-written two times, first using parallel programming on host and second time on OpenCL kernel.

3.2.3.1 Using parallel execution on the host

Two loops iterating through the columns and rows of the image were replaced by code which divided this iteration into chunks, and run each chunk on a separate thread thus breaking computation down into tasks that can run in parallel.

New algorithm used work stealing to balance a workload among available cores with the aim of increasing core utilization. If one of the cores completed its task and the other core has a long queue, work stealing would reassign tasks waiting in a queue to an idle core.

This approach allowed to distribute image processing task among all available cores but did not reduced overall computation time. The probable reason for that is that work stealing is not efficient for large numbers of processor cores. It causes significant amount of computation time to be spent in scheduling and workload balancing when running certain tasks on a multi core system. Therefore, it was decided to continue research and experimentation with other execution models. Detailed execution time profiling after implementing parallel execution by host are presented in Chapter4.

3.2.3.2 Using OpenCL kernel

As can be seen from Algorithm 3.1 new algorithm uses OpenCL write buffers to allocate image processing task to hardware accelerator.

Algorithm 3.1 ROI selection on KERNEL:

```
1:  for each image : do
2:      unsigned char image.data=OBTAIN_IMAGE_DATA(image);
3:      CL_ENQUE_WRITE_BUFFER(image.data);
4:      int row = GET_GLOBAL_ID_ROW(0);
5:      int column = GET_GLOBAL_ID_COLUMN(1);
6:      int x = ROI_X(ROI_START_X+col+ROI_START_Xtemp);
7:      int y = ROI_Y(ROI_START_Y+row);
8:      int ind =  RGB_COLOR_CHANNELS(x,y);
```

```
9:      unsigned char B = image.data[ind];
10:     unsigned char G = image.data[ind+1];
11:     unsigned char R = image.data[ind+2];
12:     FINAL_OUTPUT[INDEX] = RGB_PROCESSING(R,G,B);
13: end for
```

Line 2 processes a raw image and assigns it to a data structure of type unsigned char.
Line 3 resulting data structure is transferred to kernel using OPENCL Write Buffers
Line 4 KERNEL accesses global memory region to obtain a starting point w.r.t X axis
Line 5 KERNEL accesses global memory region to obtain a starting point w.r.t Y axis
Line 6 and 7 obtains X and Y coordinates of needed pixels
Line 8 obtains the actual index of required pixel
Line 9, 10 and 11 obtain actual values of R, G, B colors.
Line 12 Calculation is performed on RGB color channels (bit shifting, summation and etc.), the values of three channels are combined to yield the actual color of the pixel and the result is stored in a new data structure. New matrix is forwarded for further processing (grayscaling, edge detection and etc.) using OPENCL WRITE, READ and COPY Buffers.

Summary

In Botsch, 2015 this phase was executed in sequential fashion on CPU using double nested loops which was a programming bottleneck and did not allow to have multiple regions of interest due to doubling computation time with every additional ROI.

The new algorithm obtains raw image on the host and writes it to kernel. Due to ability of hardware accelerators to allocate large amount of highly parallel hardware resources, due to pipelining ability and because of the large amount of data parallelism this approach reduced computation time, contributed to more even distribution of tasks between host and kernel. Detailed execution time profiling after delegating this phase to hardware accelerators are presented in Chapter4.

3.2.4 Grayscaling

The remainder of the image (ROI) is in RGB color format. In this format each pixel contains three color channels, RED, GREEN and BLUE. In color images the values of the three channels are combined to yield the actual color of the pixel. The drawback of RGB format is that it's harder and computationally more expensive to extract lanes from the road because three color channels would have to be compared.

On contrary, grayscale image is simply one in which the only colors are shades of gray. The reason for grayscaling region of interest is that verses to colored version it needs less information to be provided for each pixel. In `grayscaled' format red, green and blue components all have equal intensity in RGB space, and therefore it is sufficient to assign a single intensity value for each pixel, as opposed to the three intensities needed to specify each pixel in a full color image [38]. There is no need to use more complicated and harder-to-process color images, grayscale images are sufficient for detecting road lane markings. Each frame is transformed to grayscale format by summing up weights of all three channels; the resulting value denotes the intensity of a pixel.

After performing grayscaling of an image road lanes could be identified based on a property that lane markings are substantially brighter than the road they are printed on, this can be seen on Figure 3.5.

Figure 3.5 Input and output of the Grayscaling phase

On Figure 3.5 each pixel reflects the intensity of the pixel in the original image and therefore dark pixel representing the asphalt will receive low intensity values and bright pixels representing white lane markings on the road will receive higher weights.
During grayscaling tens of thousands of pixels are processed independently and in parallel by hardware accelerators which provided additional performance gains. To expedite computations no floating point values were used.

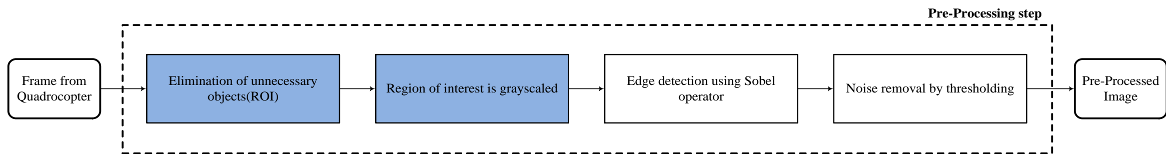


Figure 3.6: Edge Detection

3.2.5 Edge detection

Edges in an image are being detected using Sobel filter. It calculates the change in the gradient of an image and identifies regions where the frequency of color transition is higher. These regions denote sharp changes in the gradient and therefore correspond to edges in the original frame [39]

The following two 3x3 masks are used for approximating intensity gradient of every pixel. One mask is used to calculate the edge gradient w.r.t y axis, the other w.r.t x axis. Each neighboring pixel found around that point is given a value. The values are then added together and assigned to Gx andGy.

Formula 3.1 Horizontal and vertical image convolution

1. $$Horizontally\ (Gx) = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \times [dark\ image]$$

2. $$\text{Vertically (Gy)} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \times [dark\ image]$$

3. $|G| = |Gx|+|Gy|$

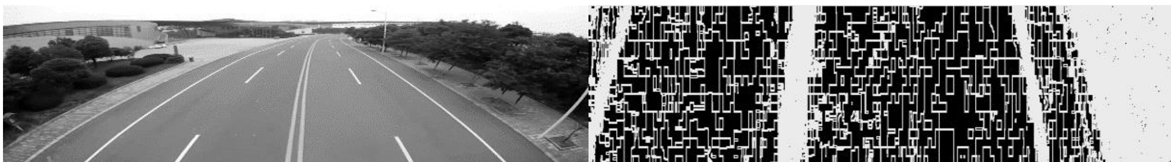


Figure 3.7 Before and after Edge detection

Line 1 - absolute value of the intensity gradient is calculated – horizontally (image convolution for horizontal direction)
Line 2 - absolute value of the intensity gradient is calculated – vertically (image convolution for vertical direction)

Line 3 - the edges are calculated using less expensive summation method: |G| = |Gx|+|Gy|

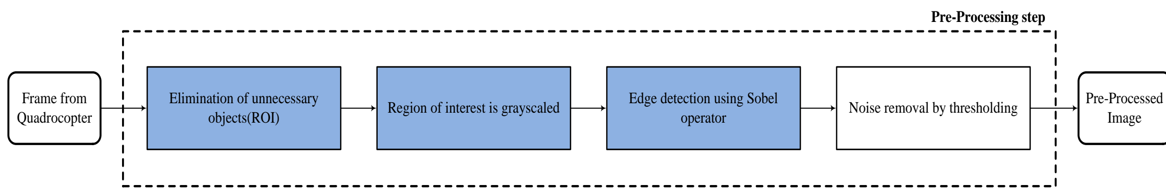


Figure 3.8: Thresholding

3.2.6 Thresholding

Thresholding is the last phase of Pre-processing. It is a method of eliminating noise from an image. In this phase value of each pixel is compared to a user defined threshold value and based on the results, the original value of the pixel gets updated by a maximum value or set to NULL.

Normally the outcome of the Thresholding phase is that regions containing lanes on a road obtain a maximum value, thus get brighter. Other disturbances such as shadows, light reflections, roadside markings and etc. are cleared off the image by assigning gradient intensity value to NULL. The algorithm is executed completely on hardware accelerators.

Algorithm 3.2 For performing Thresholding

```
1: I = INTENSITY(pixel);
2: If (I < THRESHOLD_VALUE)   then I = 0;
3: ELSE
4: I = MAX_VALUE;
```

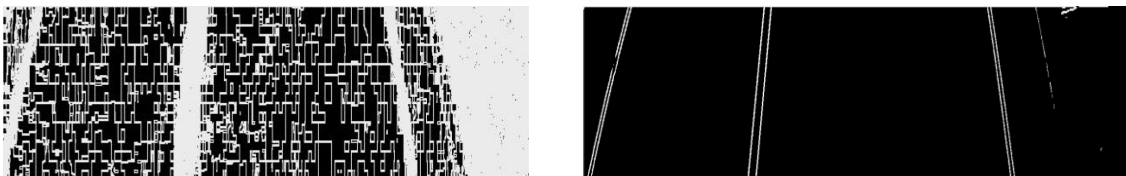


Figure 3.9: Before and after Thresholding is applied

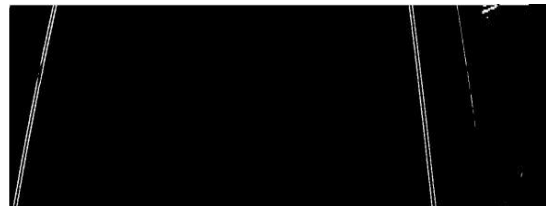
3.2.7 Summary of the Pre-processing

Algorithm 3.3 Preprocessing stage

1: SELECT_ROI (full_original_image);
2: Load required PIXELS
3: for all PIXELS P do
4: P=GRAYSCALING (P);
5: G = CONVOL_X (P);
6: Gx= Gx + G;
7: G= CONVOL_Y (P);
8: Gy= Gy + G;
9:
10: end loop
11: SUM= |Gx|+|Gy|
12: If SUM < THRESHOLDING_VALUE
13: Then SUM = 0
14: Else if SUM ≥ THRESHOLDING_VALUE
15: Then SUM = MAX.
16: End if

Line 1 – Obtains a section of the image where lanes are most likely to be located
Line 4 - Applies grayscaling
Line 5 - Obtains intensity gradient w.r.t X axis (image convolution for horizontal direction)
Line 6 – Sum of horizontal intensity gradients
Line 7 - Obtains intensity gradient w.r.t Y axis (by applying image conv. for vertical direction)
Line 8 - Sum vertical intensity gradients
Line 11- Sum of intensity gradients SUM= |Gx|+|Gy|
Line 12-16 - Clears off minor disturbances

a) Original image



b) The final outcome of preprocessing phase.

Figure 3.10 Summary of preprocessing phase

In preprocessing phase: first all unnecessary objects were eliminated by selecting portion of image where road lane markings are most likely to be found. Since even small sections of image may contain thousands of pixels eliminating some parts of the original image reduces computational effort significantly.
In the next stage the remainder of image was converted from color to grayscale. The reason for that is that verses to colored version grayscaled one needs less information to be provided for each pixel, thus reducing computational time as well.
In the third step edges were detected by applying Sobel operator.

And finally a noise is eliminated by applying Thresholding. It assigned a maximum value to pixels containing high intensity values, which most of the time correspond to lanes and NULL value to pixels containing low intensity values, thus eliminating minor disturbances from the image.

100% of the preprocessing stage is executed on hardware accelerators. Outcome of preprocessing stage is an image with lane markings of the road only. The image shown on Figure 3.10 b) is used in further phases of the algorithm for identifying the exact X and Y coordinates of lanes.

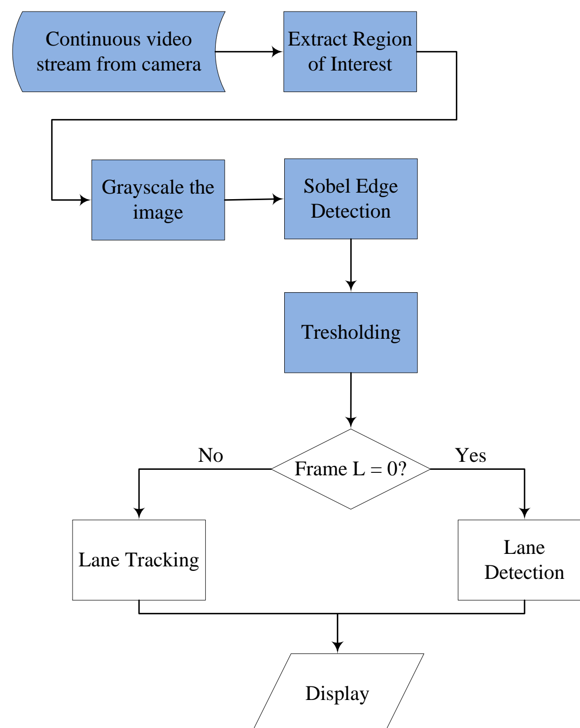


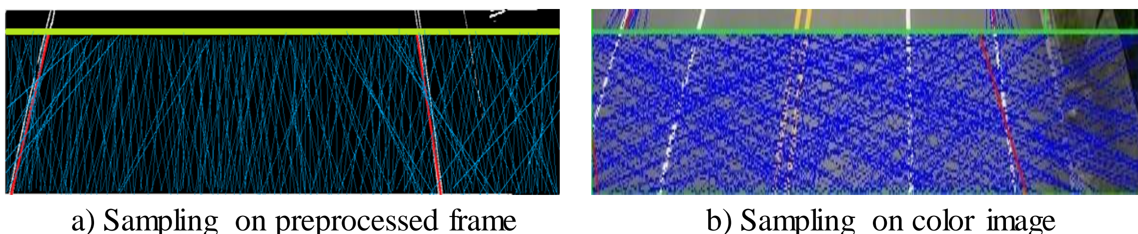
Figure 3.11: The general flow of the algorithm

3.3 Lane Detection

Detection of lane markings in image is not an easy task. That is mainly due to the poor quality of lane markings, different sorts of occlusions, presence of traffic or complex geometry of lanes. But there are few properties which make it easier to detect lanes:

- Lanes do not cross
- Each lane is located in its own region
- Regions corresponding to real lanes on average contain up to 10 pixels in width

As shown on Figure 3.11 lane detection is performed only when there is no any information available from a previous frame or lane tracking cannot proceed. As shown on Figure 3.12 Lane Detection is done by placing candidate lines randomly on preprocessed frame and calculating their weights. Weight of a line is nothing but a sum of all intensities of all pixels in the line. Thus some candidate lines correspond to real lanes, thus have a higher weight but some do not and therefore their weight is less. After candidate lines are weighted, the line with the highest weight is selected. Selected line is considered as the most accurate representation of a real line.



a) Sampling on preprocessed frame b) Sampling on color image

Figure 3.12: a) Line sampling, b) Lane Detection allocates thousands of line samples and evaluates weights of each of them

During Lane Detection step tens of thousands of sampling lines are created and weighted independently. Placing a bigger number of candidate lines produces more accurate results, but increases computational effort, especially if line sampling is done by host. In this case nested for loops would evaluate each line in a sequential manner, thus creating a programming bottleneck. But because the lines are independent from each other, this phase is executed on device.

Once sampling is completed host only performs sorting. It selects the line with the highest score, which is the most accurate representation of a real line and stores dozens of candidate lines with the weight above average inside of a data structure named good_lines which are used for lane tracking.

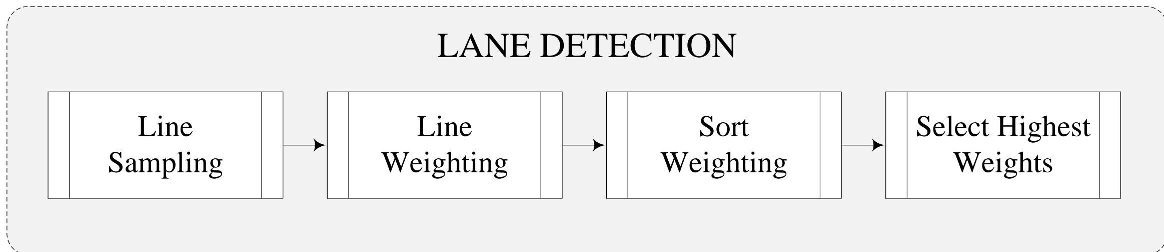


Figure 3.14: Phases of Lane Detection

As shown on Figure 3.14 Lane Detection consists of four phases and is the second most resource consuming task. On average it is triggered once for 300 frames but the amount of resources it consumes is ten times of lane tracking.

3.4 Lane Tracking

The vehicle moving at a speed of 80km/h will produce an enormous amount of data at a high speed for processing real time and it is extremely important to be able to process incoming frames on time, otherwise casualties are unavoidable. This places the algorithm into the category of hard real time systems.
As mentioned before, lane detection step is computationally expensive because it clears all the calculations done for the last image and re-processes frames afresh by allocating thousands of sample lines and calculating weight of every line, therefore, if it is performed too often, it might jeopardize the execution time, especially for frames with several regions of interests.

Benefits of Lane Tracking is that it uses pre-processed frame, good lines and best lines measurements obtained at earlier time *t-1* as an input to track the lanes in subsequent frames. Lane tracking does not detect lanes but keeps information about line markings from the previous frame, makes some adjustments to retained information based on measurements obtained real time and applies it to the current frame. It is based on Particle Filter and consists of Motion Update, Measurement Update and Resampling phases.

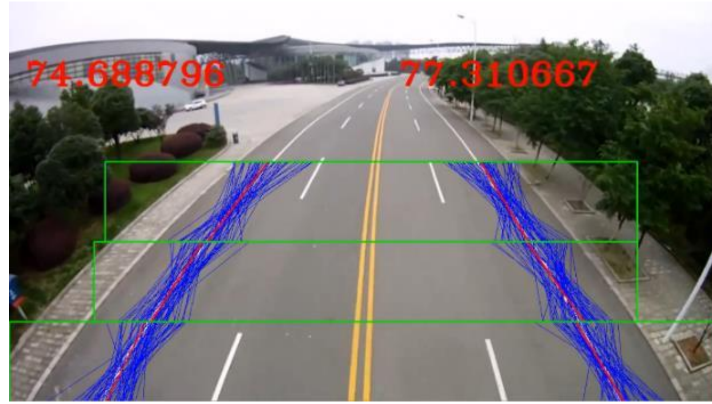


Figure 3.15: The result of lane tracking phase are exact x and y coordinates of lane markings

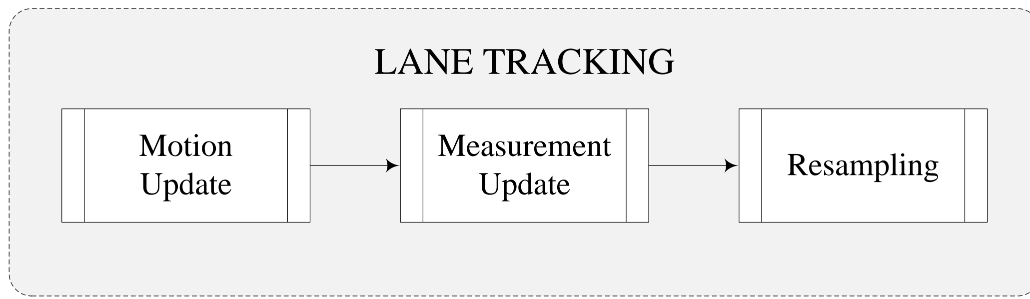


Figure 3.16: Lane tracking is performed in three phases

3.4.1 The Lane Tracking Algorithm

Algorithm 3.4 Particle Filter for Lane Tracking[40]

1: **for** $i = 1 : num_good_lines$ **do**
2: $MOTION_UPDATE\ (good_lines(i), \Delta\rho, \Delta\theta, motion_noise)$
3: $MEASUREMENT_UPDATE\ (good_lines(i), measurement_noise)$
4: **end for**
5: $*good_lines_new \leftarrow RESAMPLE\ (*good_lines)$
6: $*good_lines \leftarrow *good_lines_new$

Where i - identifies the line with the weight above average, good_lines
$\Delta\theta$ - is orientation of a line
$\Delta\rho$ – distance

Line 2 - MOTION_UPDATE - shifts each line by $\Delta\rho$ *and rotates by* $\Delta\theta$ depending on the slope and lane direction
Line 3 - MEASUREMENT_UPDATE - compares x intercept and orientation of good lines with the nearest best line and assigns a weight along the Gaussian probability distribution based on how close a good lines is to best line.
Line 5, 6 - RESAMPLING - forms a new set of particles by selecting the highest weights.

Motion Update

Motion update provides rough probability distribution of the new states X_t by using the measurements obtained one time step earlier thus the main input for motion update step are the particle sets at X_{t-1}.[20] Algorithm first constructs a temporary particle set which is very similar to the previous set but not the same. It does this by systematically processing each particle in the input set of particles X_{t-1}. The mathematical representation of the algorithm for processing particles in the input set from a previous state is based on the following formula:

Motion Update *MOTION_UPDATE* subroutine in Algorithm 3.4 is constructed based on Equation 3.1

$$Pr(X_t|Y_{t-1}) = \sum_i Pr\left(X_t^i|X_{t-1}^i, Y_{t-1}^i\right) \times Pr\left(X_t^i\middle|Y_{t-1}^i\right) \times \Delta X_{t-1} \tag{3.1}$$

Where i -is a particle
X_t – Is a new state
X_{t-1} – represents an old state
$Pr(X_t|Y_{t-1})$ – Posterior probability distribution at an earlier state

In this algorithm the state at time t is expressed as position of a lane marking with coordinates of x_top and x_bottom.

Measurement Update

Measurement update is performed after motion update to obtain more accurate probability distributions.

Mathematical representation of this stage is shown on Formula 3.2:

$$Pr(X_t|Y_t) = \frac{Pr(Y_t|X_t) \times Pr(X_t)}{Pr(Y_t)} \tag{3.2}$$

Where $Pr(X_t|Y_t)$ represents posterior probability distribution
$Pr(Y_t|X_t)$ – Likelihood
$Pr(X_t) -$ Prior probability distribution
$Pr(Y_t) -$ Evidence

Numerator of the equation (*prior x likelihood)* - $Pr(Y_t|X_t) \times Pr(X_t)$ is obtained by calculating the weight of each particle and is based on the following formula:

$$\omega_i = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\left(\frac{X_t^i - \mu}{\sigma}\right)^2} \tag{3.3}$$

Denominator of the equation, $Pr(Y_t)$ is an evidence and is obtained by summing up the weights for all the particles

$$evidence = \sum_{i=1}^{N_p} \omega_i \tag{3.4}$$

Every iteration of measurement update will filter out probability distribution of the particle states and will produce more accurate values. As a result particles located closer to actual road lane markings will be assigned a higher values compared to those far away from actual lane [20]

Resampling

Resampling or importance resampling is performed by selecting each particle by its importance weight. Resampling transforms a particle set of M particles into a new particle set by incorporating the importance weights, obtained during measurement update, into the resampling process and changes the distribution of the particles. For instance, before the resampling was performed, particles were distribution according to an earlier belief $bel(X_t)$, after the resampling particles in a new particle set are distributed (approximately) according to the posterior $bel(X_t)$ which ensures survival of the fittest by sorting out probability distribution and selecting the particles with the highest weights as shown on Algorithm 3.5. [20] The Resampling algorithm used in this paper is identical to (Madduri, 2014, p. 55-60).

Algorithm 3.5 Resample - ensures that particles re assigned to a new set have a weight less than the value of β

1: $idx = rand()\%N_p$
2: $\beta = 0.0$
3: **for** $i = 1 : N_p$ **do**
4: $\beta += rand()\%(2*\omega_{max})$
5: **while** $\beta > \omega_{idx}$ **do**
6: $\beta -= \omega_{idx}$
7: $idx = (idx + 1)\%N_p$
8: **end while**
9: $particle(i) = particle(idx)$
10: **end for**

---

idx - is a random index drawn from a particle indexes.

ω_{max} - maximum weight in the particle set

β - Variable β is assigned a random value which must be less than the double of maximum weight

ω_{idx} – weight of a particle

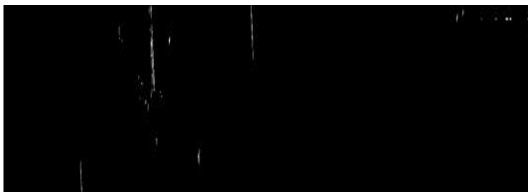
M – Number of particles in the particle set

3.5 Redetection cases

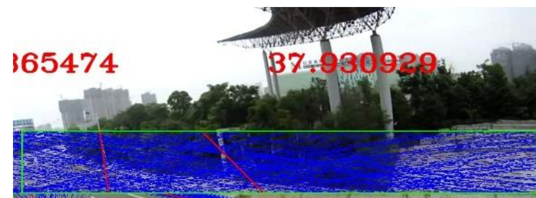
Redetection algorithm contains set of checks to ensure that lane tracking is able to proceed. It is triggered in cases where lane markings are no longer visible, vehicle moves from one lane to the other or there is significant amount of noise in the frame. Code performing the checks contains the following criteria:

- Lanes should not intersect
- The minimum distance between lines is observed
- The minimum length of lane is present in pre-processed frame– this criteria is especially important for scenarios where vehicles switch between lanes
- Lanes are not out of frame, still visible

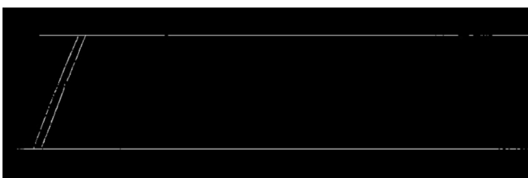
If all the criteria are met algorithm proceeds with lane tracking, if not lane detection is triggered. Figure 3.17 illustrates scenarios where redetection criteria fail



a) Lanes are no longer in the frame



b) Redetection is triggered immediately



c) Vehicle switches between lanes, thus only part of lane is in the frame

d) Initialization of the algorithm

Figure 3.17 Redetection cases

Lane detections are mostly triggered during lane changes, for first frames, presence of significant amount of noise in the frame or when line is no longer visible.

3.6 Angle calculation

For the calculation of the inclination angle, the connection between two or more regions of interests (ROIs) is necessary. Weather two or more ROIs are connected is computed in the following way:

- Right and left side (X_{start}, X_{end}) coordinates for each ROI are stored
- If X_{end} coordinate of ROI one is in the range of

  $$X_{start_ROI_0} - predefined_offset < X_{end_ROI_1} < X_{start_ROI_0} + predefined_offset$$

  it is assumed that these two ROIs are connected

- after connection is defined next ROI's connection should be detected in the same way
- If all ROIs are connected, the flag value will change to true. This indicates that all ROIs for the current frame form one line, as shown on Figure 3.18
- Calculation of inclination angle is performed
- The same procedure is applied to the right side of the ROI

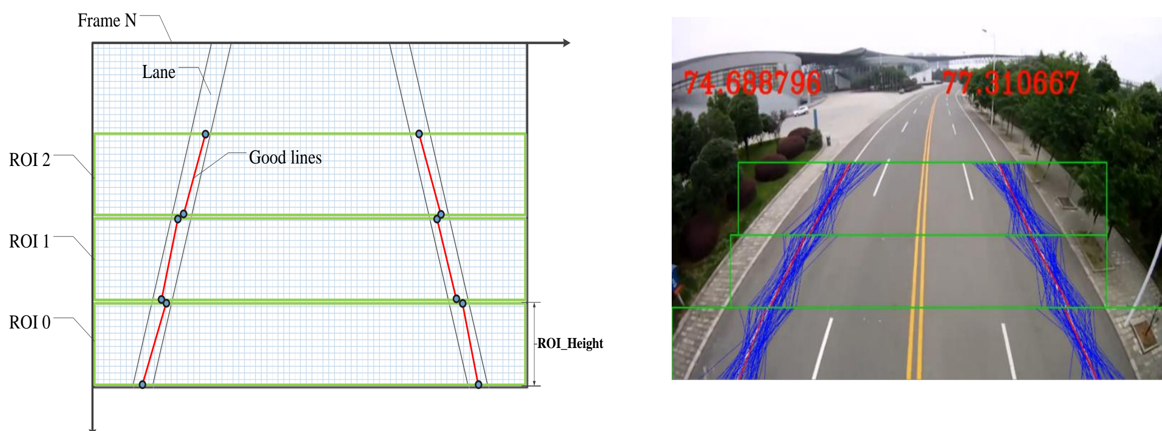


Figure 3.18 Best lines across all ROIs approximately form one line

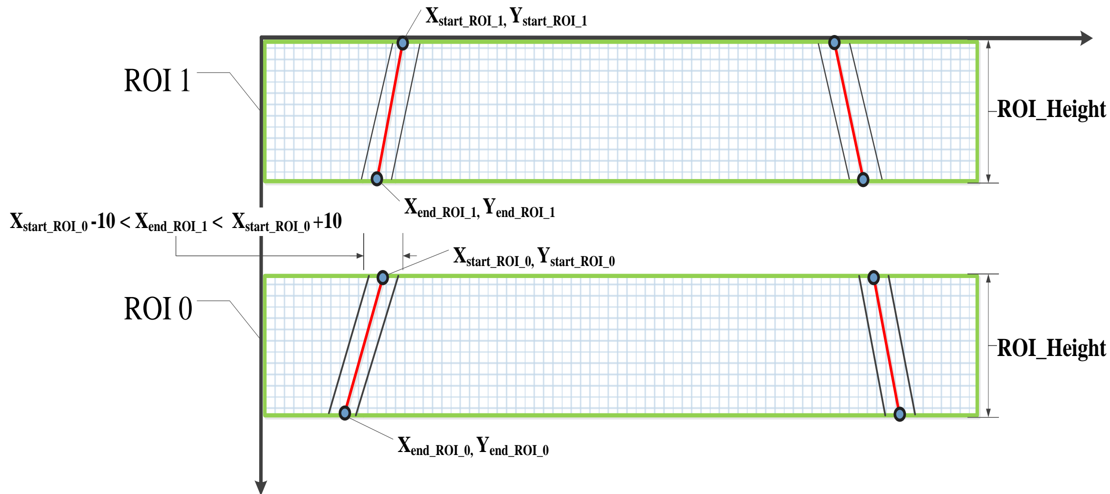


Figure 3.20: Scheme of Connection detection between ROIs

The following formula is used for calculating the inclination angle:

$$\tan^{-1}\left(\frac{a}{b}\right) \tag{3.5}$$

where a, b - cathects of the rectangle shown on Figure 3.21

a is height of one ROI multiplied to the number of all ROIs

b is equal to difference of X_{start} coordinate of the last ROI and X_{end} coordinate of the first ROI.

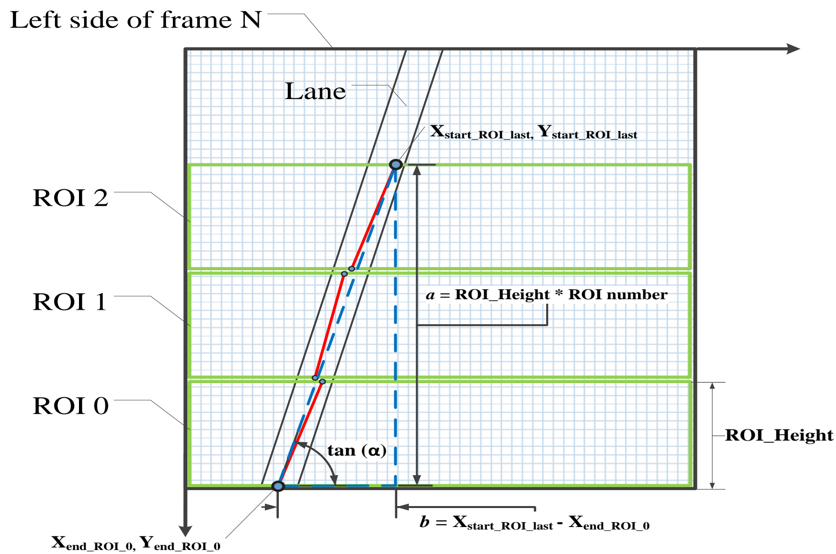


Figure 3.21: Scheme of calculation of inclination angle

As can be seen from Figure 3.18 after computing inclination angle, the value is printed on a frame.

3.7 Further contributions

3.7.1 Multiple regions of interests

In order to provide a long distance sensing of lane markings it was suggested by Botsch, 2015 to populate frame with multiple, independent ROIs. To be able to achieve that, some sections of code in pre-processing phase had to be replaced.

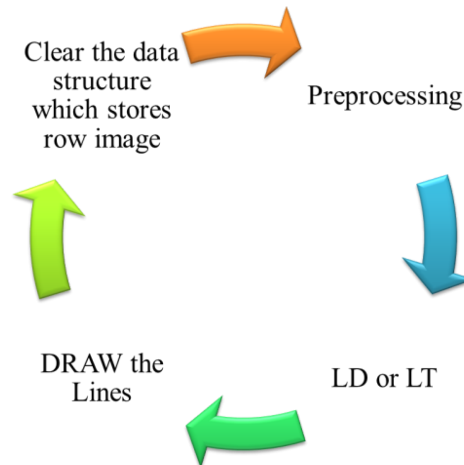
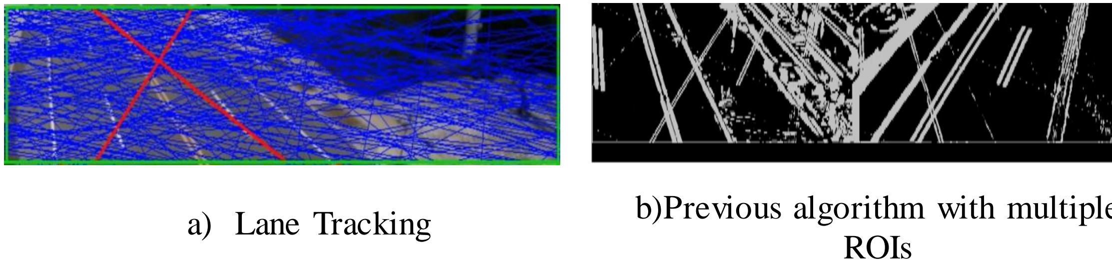


Figure 3.22: The process of processing ROIs

For instance, in order to draw multiple ROIs on the frame, it had to pass through preprocessing phase several times, but with different *image_width*, *X* and *Y* coordinates thus processing different sections of image. It was not possible with the previous algorithm because previous algorithm was designed to receive image, detect/track lanes and dispose the image. Therefore when the frame did not get disposed but went through preprocessing phase multiple times the algorithm wrote one frame on top of the other, creating several layers of information as shown on Figure 3.23. For example, ROI_0 from Figure 3.20 was preprocessed and its values were stored inside of *image_raw* data structure. In the next iteration when the same frame was passed through preprocessing phase again but with different ROI coordinates, algorithm wrote new values on top of previous *image_raw*. As a result final preprocessed frame had several layers of information thus was corrupted and the rest of the algorithm did not function any more, Figure 3.23a.



a) Lane Tracking

b)Previous algorithm with multiple ROIs

Figure 3.23 a) Lane Detection/tracking on corrupt frames b) Preprocessed frame with several layers of information

The problem was solved by "cleaning" the specific memory regions inside of *image_raw* data structure before using it in preprocessing phase again. In a new approach *image_raw* stores values of ROI_0 first, than on the second iteration left and right edges of *image_raw* are set to 0 and values in the middle are replaced by new measurements obtained from ROI_1. The values which are set to 0 are filtered out by thresholding phase, therefore do not affect the results of Lane Detection/Tracking.

3.7.2 Adaptive regions of interest

One of the aims of this work is to achieve an accurate and fast lane detection/tracking for multiple regions of interest for views from top and front Since, *"The size of the ROI is a driving parameter that determines computational speed and effort of the lane detection/tracking. The smaller the ROI is chosen, the faster the lane detection performs"* - *Botsch 2015*. *Therefore* straightforward allocation of ROIs on a frame would linearly increase computation time, and algorithm would not be able to meet hard real time requirements. Adaptive (meaning that the *WIDTH*, *X* and *Y* coordinates of *ROI's* are not hardcoded but keep adjusting w.r.t. lanes) *ROIs* guarantees that the *image* to be processed is as small as possible and mainly focused on road lane markings.

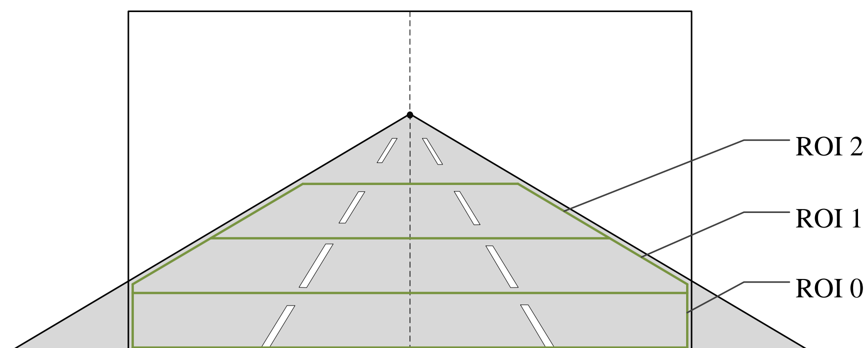


Figure 3.24 Adaptive ROIs

"Adaptivity" was achieved by dynamically defining *ROI WIDTH* based on *X* and *Y* coordinates of lanes detected in a previous ROI. As shown on Figure 3.20 width of *ROI_1* is smaller than *X_END_ROI_0* thus maximum width limit set for *ROI_1* can be calculated based on coordinates of lanes detected on *ROI_0 and so on*. Execution time profiling presented in subsequent chapters illustrates that this feature slightly reduced computation time.

3.8 Summary

Programming Bottlenecks: There were several code segments across the project where computations were performed neither on device nor in parallel. As shown on Figure 3.4 *Distribution of computation time between host and device,* bulk of the computations were performed by host, while hardware accelerator was idle most of the time. This was one of the areas which required careful analysis. After several weeks of research more even distribution of the workload across hardware accelerators and the host was achieved and had a positive impact on overall performance. It was accomplished by eliminating selection of ROIs in a sequential manner by host and delegating this task to kernel. This allowed breaking computation down into tasks that can run in parallel by hardware accelerators which gave shorter computation time and more even distribution of workload between host and kernels, more about it in Chapter 4. Currently 100% of pre-processing is done by kernel. This allowed achieving maximal computation speed. It was also calculated that communication overheads for transferring original image to kernel and back to host are insignificant.

The next contribution is developing an algorithm for multiple ROIs and lane inclination angle calculation: There was a suggestion to develop an algorithm which would be able to compute inclination angle of the lane. *Which would allow* DAS to deal with road bending and sharp turns in advance. The developed algorithm detects connection between all best lines on multiple regions of interest, computes the length of all sides of an imaginary triangle and calculates the degree of inclination. Allocation of multiple ROIs was achieved by passing frame through replaced preprocessing, updated lane detection and updated lane tracking phases multiple times.

*And finally a*daptivity for ROIs was achieved by dynamically changing ROI width and *x_start* values based on lanes detected in other ROIs. The computational time gains after implementing adaptivity feature are presented in the following chapter.

All above mentioned changes contributed towards more feature reach lane detection/tracking algorithm, resulted in more even distribution of workload between host and kernel and allowed to shorten computation time.

Chapter 4

Results

4.1 Chapter outline

Testing of the algorithm was conducted to ensure that planned goals have been achieved and defects or sections of code degrading performance were found and fixed.

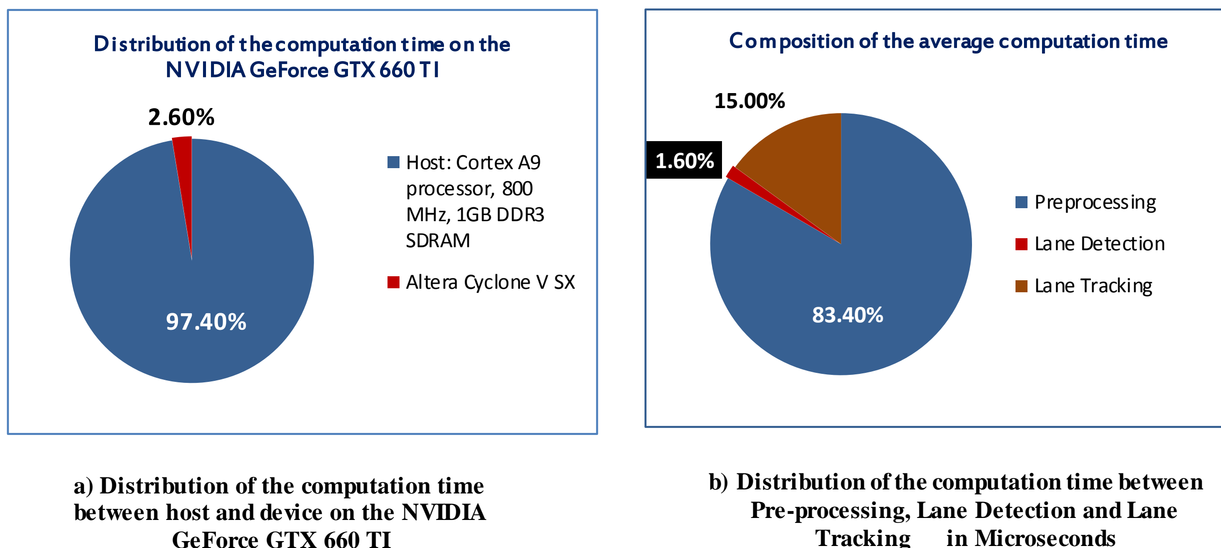
This chapter is structured the following way, in section 4.2 computation time and distribution of workload between host and device in Botsch 2015 and in current work is shown. Section 4.3 presents testing of the algorithm on TUM_DLR dataset and compares execution time obtained with adaptive ROIs to measurements obtained with fixed sized ROI. The testing in section 4.3 was conducted via allocating different number of particles and defining different value for a threshold parameter. In the last section some know issues were presents.

For evaluation of the performance of the algorithm testing was performed on TUM_DLR dataset which was recorded during day time in the surroundings of German Aerospace Centre, Munich and TUM_DAY dataset, recorded in the surroundings of Garching.

4.2 Composition of the computation time.

4.2.1 Initial Performance

Initially the algorithm of Botsch 2015 was evaluated against TUM_DLR dataset by allocating one ROI. Profiling revealed that pre-processing phase alone occupied more than 80% of execution time and the remaining phases like Lane Detection and Lane Tracking took only 1.6% and 15%. This is shown on Figure 4.1b)
Also on Figure 4.2 it is shown that within preprocessing stage 80% of the computation time is spent for selection of ROI and 20% of time is spent for Grayscaling, edge detection and Thresholding. Besides that as shown on Figure 4.1a) distribution of the workload between host and kernel is 97.40% and 2.60% respectively. The distribution is uneven with host performing bulk of computations.



a) Distribution of the computation time between host and device on the NVIDIA GeForce GTX 660 TI

b) Distribution of the computation time between Pre-processing, Lane Detection and Lane Tracking in Microseconds

Figure 4.1 Distribution of workload and computation time in Botsch 2015

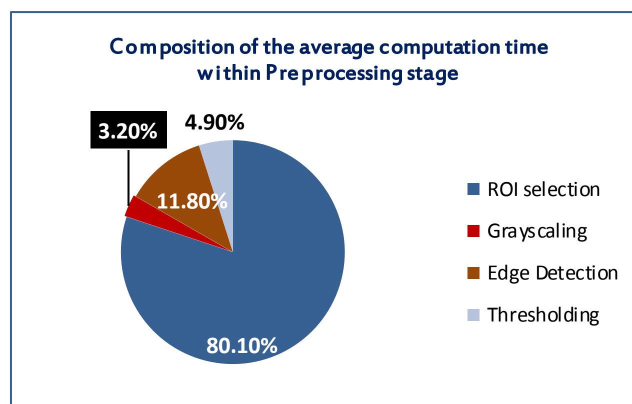
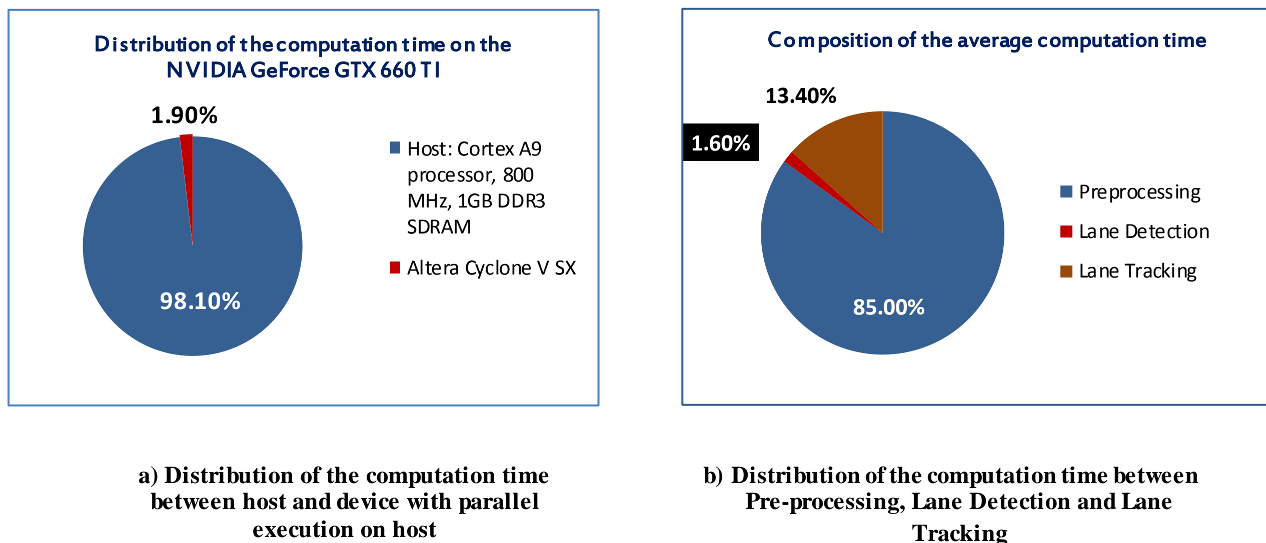


Figure 4.2 Distribution of the computation time in Botsch 2015

4.2.2 Parallel processing by host

As mentioned in Chapter 3, initial attempt to improve computation time was done by introducing parallel for, in hope that it would break computation down into tasks that can run in parallel. If computational results are satisfactory, there will be no further need in developing OpenCL kernels. Thus segments of code occupying bulk of computation time due to sequential execution on host could still remain on host but instead would be executed in parallel. This would shorten development time, as writing kernel code, managing memory problems and integrating kernels into the main application is a time consuming task. The outcome of this attempt is presented below.

Parallel for allowed to distribute image processing task among cores but did not reduced overall computation time. The measurements obtained after implementing parallel execution on the host are shown on Figure 4.3 and Figure 4.4. From Figure 4.3b) it can be seen that pre-processing phase occupied 85% of execution time and the remaining phases took 1.6% and 13.4%. Also on Figure 4.4 it is shown that within preprocessing stage more than 81% of the time is still spent for ROI selection, distribution of the workload between host and kernel is even more skewed towards host showing 98.10% and 1.90% respectively where host still performs bulk of computations



a) Distribution of the computation time between host and device with parallel execution on host

b) Distribution of the computation time between Pre-processing, Lane Detection and Lane Tracking

Figure 4.3 ROI selections on the host with parallel loops

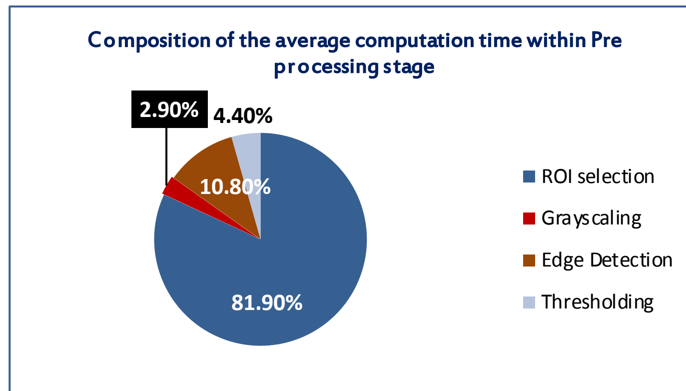
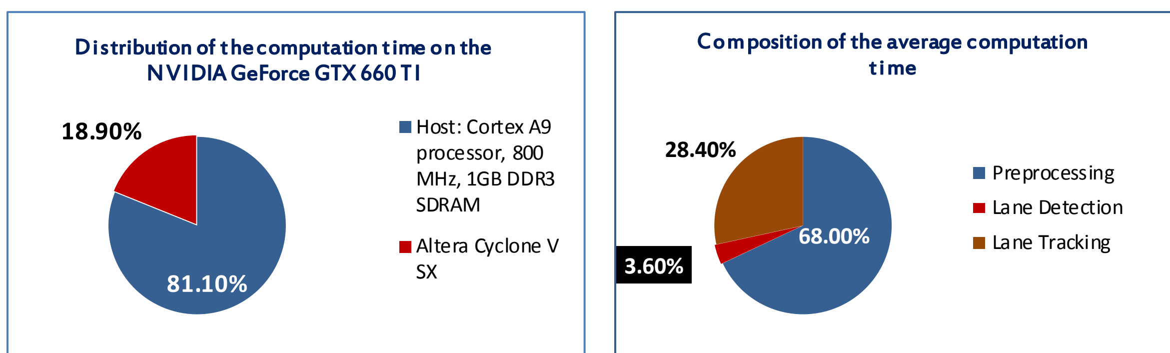


Figure 4.4 Distribution of computation time with parallel loops

As can be seen on Figure 4.3b and Figure 4.4 Pre-processing took a bit more time than the previous approach.

4.2.3 Parallel processing by kernel

In this approach the idea was to delegate most resource consuming tasks to hardware accelerators. After allocating ROI selection task to the kernel the following measurements were obtained.



a) Distribution of the computation time between host and device on the NVIDIA GeForce GTX 660 TI

b) Distribution of the computation time between Pre-processing, Lane Detection and Lane Tracking in Microseconds

Figure 4.5 ROI selections on kernel

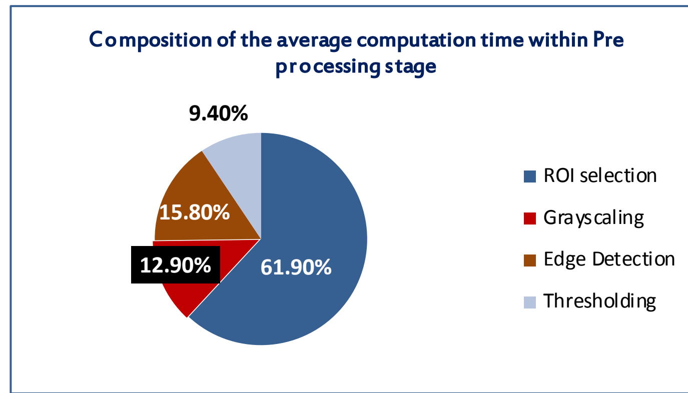


Figure 4.6 Distribution of computation time within Pre processing

The new algorithm obtained raw image on the host, and delegates it to the kernel, where all computations are performed. This approach reduced computation time and contributed to more even distribution of tasks between host and kernel. Communication overheads incurred during write and read operations are negligible.
As can be seen from Figure 4.5b) hardware accelerators handle image processing tasks more efficiently, pre-processing phase occupies less than 70% of execution time. Also on Figure 4.6 it is shown that within preprocessing stage less time is spent for ROI selection. Distribution of the workload between host and kernel is more balanced, with 81.10% and 18.90% respectively.
The host still consumes significant amount of computational time (more than 81%) due to ARM Cortex-A9 processor running on the host and not by the programming bottlenecks in the algorithm or the Altera FPGA board and also execution time profiling revealed that significant amount of computation time is spent for loading images and writing them back to the disk.

4.3 Testing the algorithm on datasets

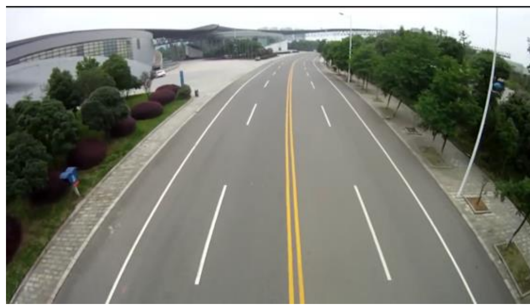
Following computation time profiling the algorithm was tested on pre-recorded video which was captured during daytime in the surroundings of German Space Operations Centre. The dataset mostly contains the suburban roads with less traffic and is suitable for testing the algorithm with multiple ROIs.The video was recorded by Parrot Bebop Drone quad copter.

Recorded Video	TUM_DLR	Name of the dataset
ROI1	1280	Size of the first ROI
ROI2	800	Size of the second ROI
ROI3	680	Size of the third ROI
NUM_FRAMES	686	Total number of frames in the dataset
THRESHOLD_VALUE	100,150,200,250,300	Only weights above threshold value is considered
GOOD_LINES	16,32,64,128,256	Number of particles used for Lane Tracking

Table 4.1 Testing settings - the algorithm was tested with different parameters for threshold and sampling lines.

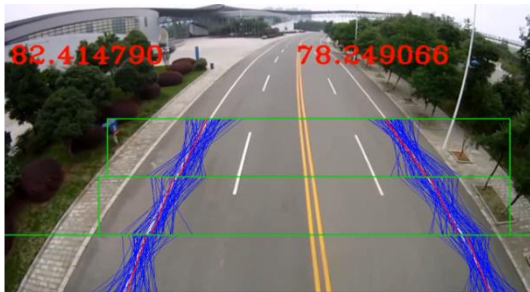
4.3.1 Lane Detection

On Figure 4.7a) an image with clearly identifiable lane markings is shown. Lane detection algorithm uses the same frame to allocate multiple ROIs and detected the lanes using 128 sampling lines, the outcome is shown on Figure 4.7c).
From Figure 4.7b), it can be noticed that the lane markings occupy the entire ROI and have different orientations. This was done to handle the cases where lanes are not straight but have complicated road lane geometry.
Following lane sampling the algorithm selects the two best lines, to represent the real lanes. As can be seen from Figure 4.7c) d) selected two best lines are the perfect match to the real lane. These experiments were conducted to prove that the algorithm is able to detect/track lanes for images taken from cameras installed on top of the vehicle across multiple ROI.



a) Full original image

b) Allocation of hundreds of sampling lines



c) Multiple ROIs are allocated

d)Detected best lines match the real lanes

Figure 4.7 b) Orientation of sampling lanes are in all directions and cover the whole ROI. c) Lane detection/tracking is performed separately for each ROI and is not influenced by neighbouring ROIs across all frames in the dataset. d) The accuracy of detecting the lanes in the algorithm is high for 256,128 or 64 sampling lines but degrades if number falls below 64.

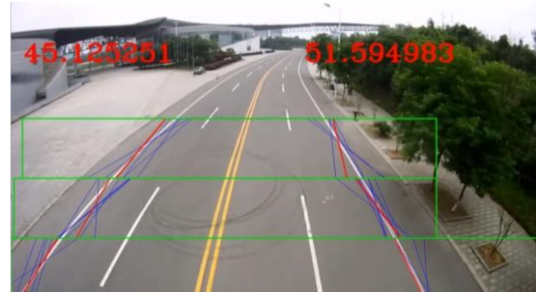
4.3.2 Lane tracking

As mentioned in previous sections lane detection is triggered on average once for 300 frames (depending on the quality of the recording and lane geometry). It is triggered mostly during lane switches, intersection of detected lanes or if minimum distance between lanes is not observed. The rest of the time lane tracking operation is performed. Therefore performance of the algorithm is heavily dependent on the performance of lane tracking phase.

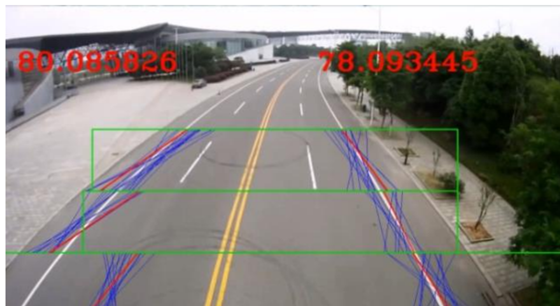
Performance of Lane tracking phase is dependent on the number of allocated particles and ROI size. Higher are the numbers more time it takes for the algorithm to compute the results.

Sections below presents testing conducted on the dataset with different number of particles. The purpose of this testing was to find the minimum number of particles required for obtaining accurate results.

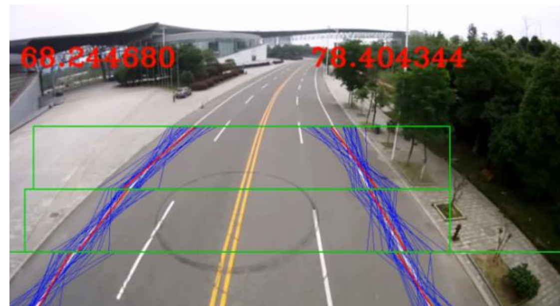
a) Original image



b)Number of particles is set to 16



c) **32 particles**



d) **64 particles**

e) **128 particles**

f) **256 particles**

Figure 4.8 Lane tracking/detection performs well for 256, 128, 64 sample lines but allocating a number less than 64 degrades the accuracy of the algorithm.

Figure 4.8, presents the results obtained for different number of particles. Figure 4.8a) shows the original image and images 4.8b) to 4.8f) show different outputs using 32, 64,128,256 particles. The outcome of the testing is that the algorithm is able to track lanes with 32 particles, but not across all ROIs. If second or third ROIs

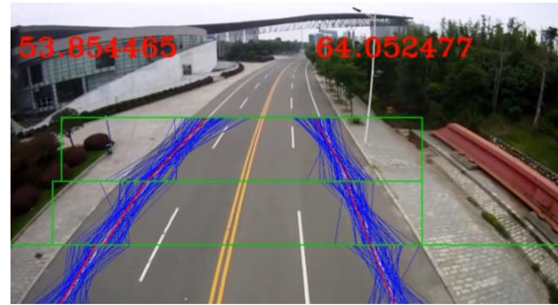
will have complicated lane geometry or weights of lanes will be less then weights of white buildings in the vicinity the Particle Filter might not be able to perform tracking any more. This will trigger lane detection more often and will result in increase of overall computation time. Defining the number of particles to a bigger value (64, 128, and 256) generally improves the accuracy of the algorithm but requires additional computation time.

In summary: The robustness of the lane tracking algorithm was evaluated across all ROIs for all frames in the dataset. For 98% of the frames algorithm was able to track lanes accurately when 64 particles were allocated, this can be seen from Figure 4.8d).

4.3.3 Threshold

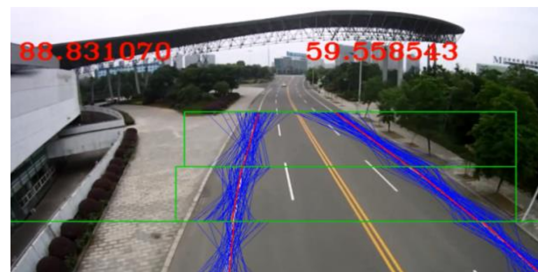
As mentioned before the **TUM_DLR** dataset was tested with different threshold values. The testing was conducted to find out the minimum required value for this parameter.

a) Full original image

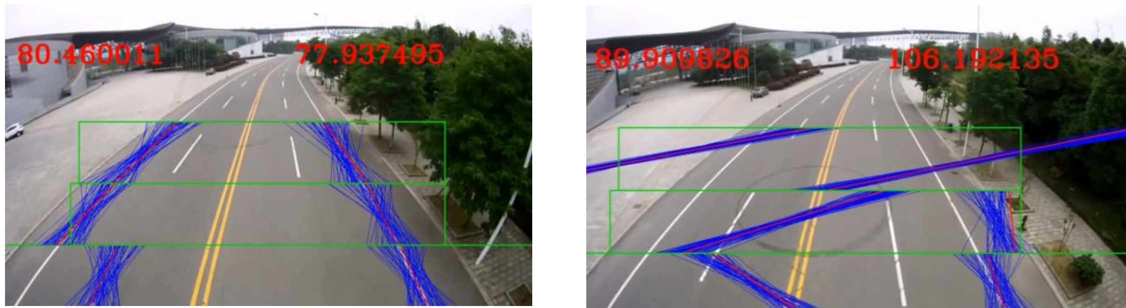


b)Threshold value is equal to 100

c)For 96.2% of the frames the algorithm produced accurate results when threshold value is set to 150



d)Threshold value is equal to 200



e) Threshold value is equal to 250

f) Threshold value is equal to 300

Figure 4.9 testing performed on TUM_DLR dataset with various threshold values

Testing revealed that the algorithm produces optimal results across all ROIs when the threshold value is set to 150.

4.3.4 Adaptive ROIs

Experiment was performed on a TUM_DLR dataset with three ROIs to measure computation time of the algorithm *with* adaptive ROIs. Following the experiment the results were compared to measurements obtained when “adaptivity” feature is disabled.

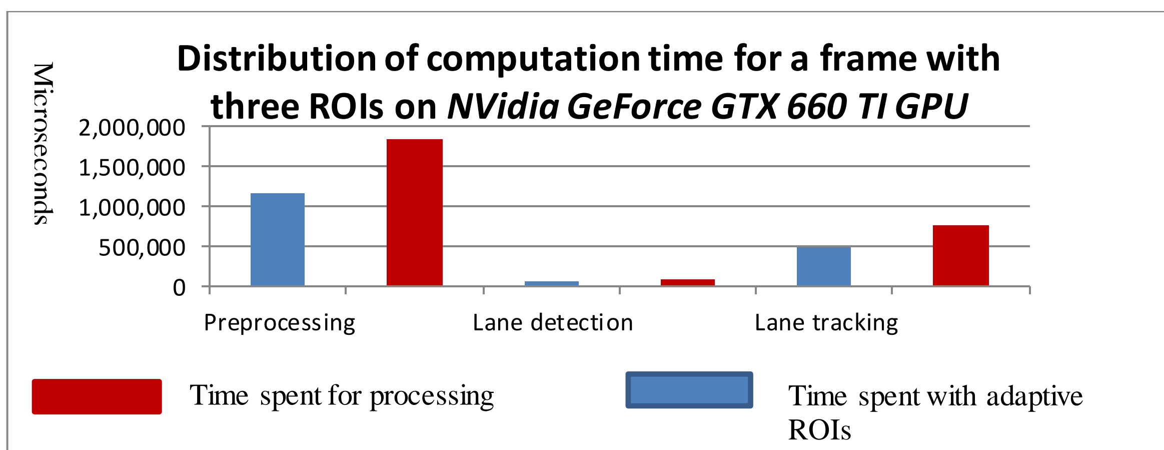


Figure 4.10 Algorithm with adaptive ROIs is computationally faster

The size of the ROI has a great influence on computational speed of the algorithm. The smaller the size of the ROIs, the shorter is the computation time. Adjusting ROI size dynamically for a frame with three ROIs reduces computational time for preprocessing phase for 30%, lane detection/tracking phases for 35%.

Table 4.2

a) Distribution of computation time for a frame with three non-adaptive ROIs

Operation	**Time in microseconds**	**Frames per second**
Preprocessing	1.844.640, 00	
Lane detection	86.232,00	
Lane tracking	767.286,00	
Total	2' 698 '158	*114*

b) Number of processed frames per second for a frame with three non-adaptive ROIs

Number of particles	**Frames per second**
64	114
128	123
256	123
512	117
1024	105
2048	88

Table 4.3

a) Distribution of computation time for a frame with three adaptive ROIs

Operation	**Time in microseconds**	**Frames per second**
Preprocessing	*1.148.880*	
Lane detection	53.969,00	
Lane tracking	475.994	
Total	*1' 678 '843*	*194*

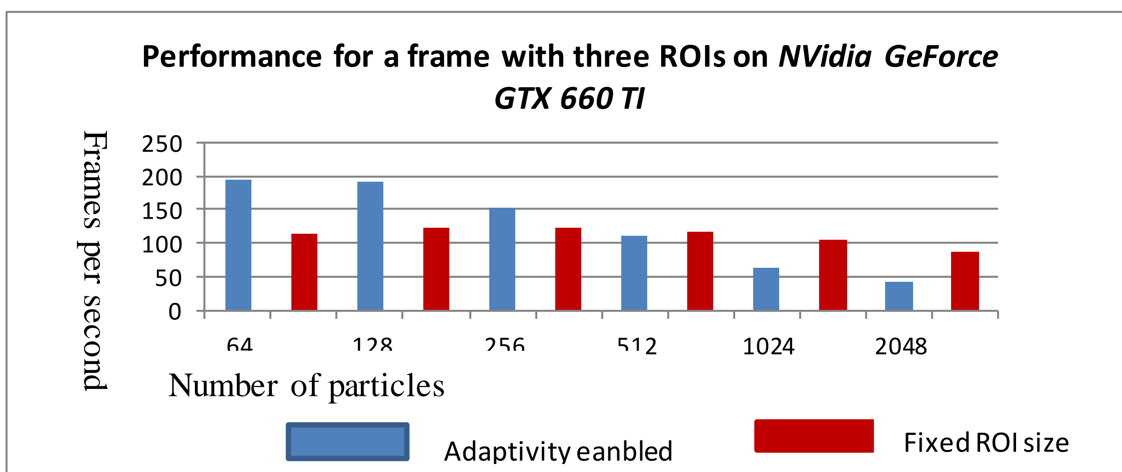
b) Number of processed frames per second for a frame with three adaptive ROIs

Number of particles	**Frames per second**
64	194
128	190
256	151
512	111
1024	64
2048	41

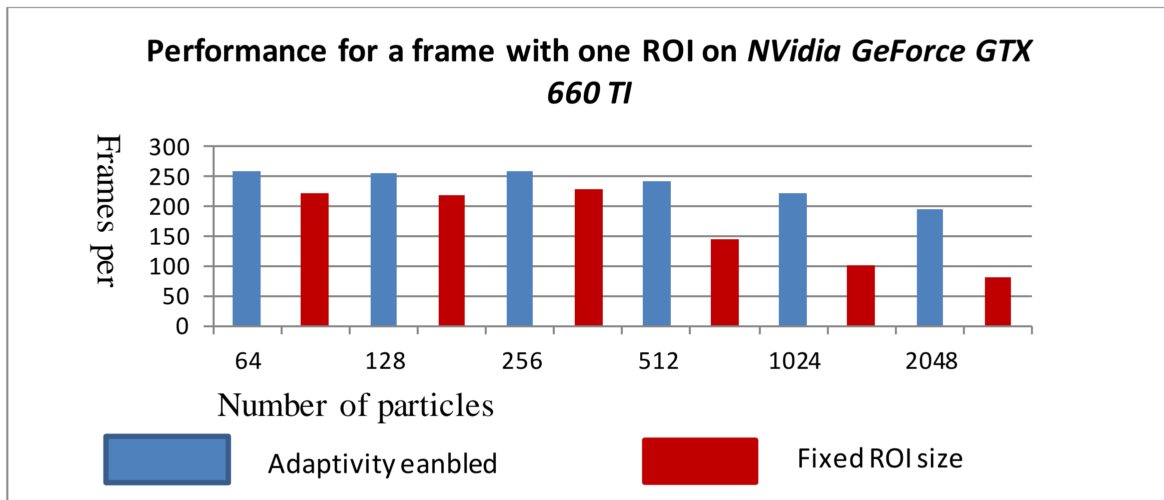
4.3.5 Computation Speed

As mentioned earlier the algorithm developed in this thesis falls under a category of hard real time systems, and thus should detect/track lanes as fast as possible, therefore computation speed and accuracy are the most important characteristics of the algorithm.

Number of processed frames and computation time are dependent on few factors like, the size of the ROI, the number of ROIs and number of particles. Figure 4.11a) and b) demonstrates the influence of those factors.



a) *Diagram above depicts measurements obtained. Y axis represents number of frames per second processed on GPU, X axis shows allocated number of particles. During the experiment three ROIs were allocated for each frame. As can be seen from diagram above performance is higher for 64, 128 and 256 particles when "adaptivity" feature is enabled. But if more particles are allocated algorithm performs better when ROI size is fixed.*



b) *During the experiment one ROI was allocated for each frame. The results are different, the performance is higher for 64, 128, 256, 512, 1024 and 2048 particles when adaptivity feature is enabled, and algorithm performs better when ROI size is fixed if less particles are allocated.*

Figure 4.11 Performances on GPU

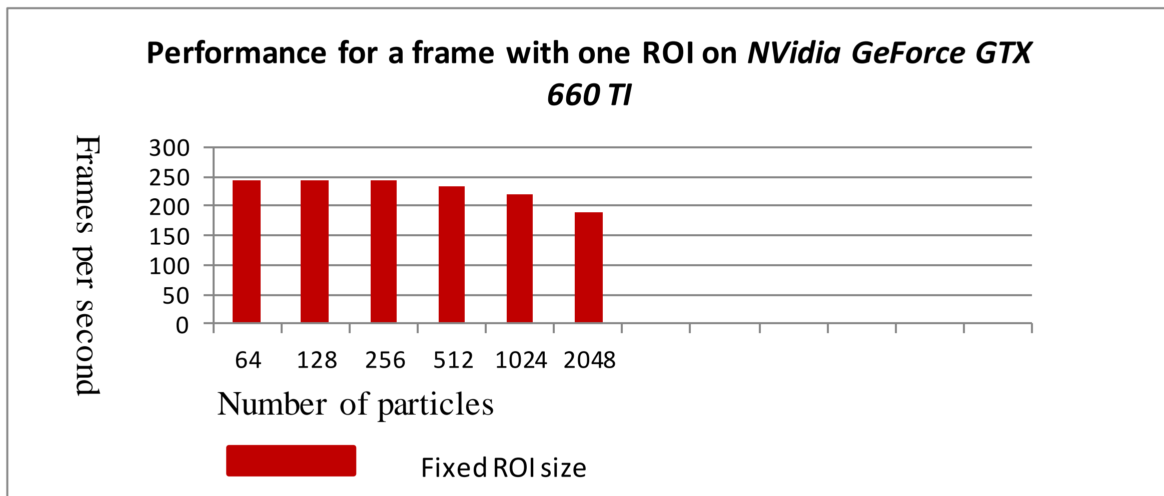


Figure 4.12 Performances on GPU of Botsch, 2015

Number of particles has major influence on the processing speed. On GPU the average frames per second decrease by less than 80 % if the number of particles is raised from 64 to 2048 for three ROIs and decreased by less than 25% for frames with one ROI.

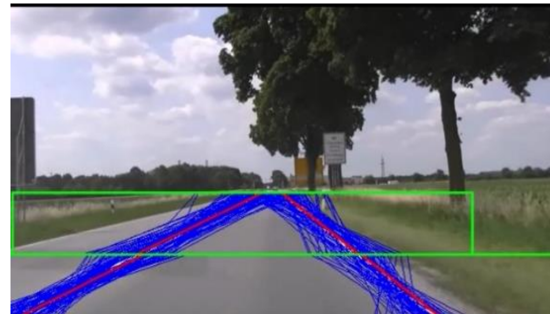
Current algorithm produces accurate results with 64 particles. With 64 particles on average 258 frames per second are processed for one adaptive ROI, and on average

194 fps are processed for frames with three adaptive ROIs. The results show the slight increase in performance compared to a previous work.
In Botsch 2015, on average 245 fps are processed with 64 particles (excluding inclination angle calculation) versus to 258 fps with the same number of particles in current work (including angle calculation)

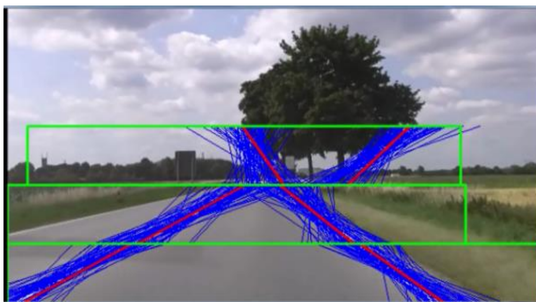
4.4 Known problems

Testing revealed that algorithms working for one scenario may not work well in others. Examples are the datasets like TUM_DAY where camera is installed in front of the vehicle and has a view directed towards the horizon. In this case detection/tracking of lanes will be accurate for the first ROI but will fail for the rest because other ROIs are not visible. But for cases where frames are captured from quad copter or from a camera installed on top of the vehicle, ex. TUM_DLR dataset, the algorithm is able to detect/track lanes across all ROIs.

a) Algorithm works with one ROI



b) Lanes on a second ROI are still visible



c) Lanes on third ROI are no longer visible

d) Tracking/detection for two ROIs

e) Tracking/detection for three ROIs

Figure 4.13 a), b) For frames taken by a camera installed in front of the vehicle the algorithm will detect/track lanes accurately only on two ROIs. c) Within the third ROI lanes no longer exist, d), e) the camera should be installed on top of the vehicle in order to detect/track lanes for higher number of ROIs.

4.5 Summary

Initial section presented issues of the previous work and different approaches taken to solve them. The most efficient strategy to reduce computation time was to delegate image processing task to hardware accelerator and read results back to the host using Open CL read, write and copy buffers. This approach allowed to decrease ROI selection time by 20%.
In subsequent sections the testing of the algorithm against different datasets was presented along with sample outputs and obtained computation times. The algorithm was tested against TUM_DLR and TUM_DAY datasets with the help of shell script. The script triggered the program several times and on every fifth execution increased number of particles [64, 128, 256, 512, 1024, and 2048]. The final measurements representing computation time, processed frames per second, distribution of the workload between host and device were calculated by summing up the intermediary results and obtaining an average value. Obtained measurements were compared to the initial results and conclusion was drawn. In the last section of the chapter some known issues were presented.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The algorithm consists of three main phases:

- preprocessing
- lane detection/lane tracking

During preprocessing phase original image is first cleared of all unnecessary objects (ex. Sky, trees and buildings) by selecting the region of image, ROI – where lane markings are most likely to be located. Selected region is grayscaled, mainly for practicality reasons, applied Sobel filter and cleared from minor disturbances.

Execution time profiling showed that significant amount of computation time was spent on preprocessing stage therefore this phase has been replaced by kernel and delegated to hardware accelerators. As a result 100% of preprocessing is now performed by kernels. This allowed accomplishing more even distribution of the workload between device and the host. Communication overheads for transferring image data between host and device are insignificant.

Following the preprocessing phase, either lane detection or lane tracking is performed. In case of lane detection – region of interest is populated with hundreds of random sampling lines to sample probable road lane markings, lines are weighted according to their distance to the lane and line with the highest weight is selected to represent the real lane.
In case of lane tracking – the weightings from a previous frame, motion noise and measurement noise are used to track the road lane markings in the current frame. Lanes are not evaluated again but are tracked. 100% of lane detection and lane tracking are also computed by hardware accelerators.

Previous algorithms were designed for one ROI: ROI was extracted, preprocessed, passed through lane detection/tracking and disposed. New algorithm is able to support multiple, independent, adaptive ROIs. Adaptivity was accomplished by dynamically changing ROI width and x_start coordinates.

The inclination angle of detected lane is calculated and printed on an original image. Thus, enabling on board system to deal with road bending and sharp turns in advance. Calculation of inclination angle was done by detecting connection between all best lines across all ROIs, and applying trigonometric functions to obtain the degree of inclination.

The algorithm was tested on datasets with varying conditions, lane types and camera inclination angles. The results showed that overall computation time was reduced by 15% and more even distribution of workload between host and kernel was achieved. The current computation time ($t > 85\%$) consumed by ARM Cortex-A9 processor is justified due to peripheral tasks, read/writes performed by host and not by the programming bottlenecks or design faults in the algorithm.

Testing revealed that the computation time differed with the selection of the ROI size, number of ROIs and number of particles. The algorithm was tested on Nvidia GeForce GTX 660 TI GPU and compared to preceding work. The results showed a slight increase in accuracy and robustness, approximately 15% faster execution on the GeForce GTX 660 TI, due to elimination of programming bottlenecks and implementation of parallel execution on kernels.

All above mentioned changes contributed towards feature reach, accurate and robust lane detection/tracking algorithm.

5.2 Future Work

The proposal for future work is drive repetitive routes autonomously. Repetitive routes are the routes taken by drivers often; ex. Is the road from home to office and back driven at least five times a week (~20 hours a month).

During the supervised learning phase the algorithm uses training sets (recorded stereo images from on board cameras), to identify lane markings, objects, patterns and match them to actions taken by the driver. Obtained information is used as an input to learning algorithm. Next time the algorithm is used it can output the same steering action for a similar occurrences. Initially accuracy may be low, but as algorithm learns the driver can rely on the vehicle to drive autonomously from home to office and back at least five times a week. Technology developed in this thesis can serve as a good starting point.

In order to achieve semi-autonomous driving it is suggested to:

1) First, model a set of virtual routes of different complexity, including a straight road, a circular route, a route with multiple turns, etc. Given certain amount of manually designed road types, it is possible to utilize them as building blocks for generating a comprehensive set of routes that could be encountered in real world.
2) Draw virtual lanes across these roads, thus obtain ground truth.
3) Model a virtual car equipped with a set of several cameras and possibly some other sensors such as LIDAR.
4) Drive the car along these routes and record the videos from the cameras. With recent advancements in rendering technologies, it is possible to obtain very realistic videos.
5) Using the ground truth of the lane, supervised learning (e.g. a convolutional neural network) to train a model to recognize the lane from video records and match it to actions taken by driver.
6) After achieving this task it will be necessary to include traffic movement, cyclists, and pedestrians. Afterwards, the model needs to be adapted to dynamic and stochastic environments. The final goal of applying supervised learning is to achieve an automated driving experience for one single route. Due to threat imposed on lives of people the reliability of the recognition is of particularly high concern for such systems. A reinforcement learning method that supports successful driving experiences (e.g. identifying and following the lane without accidents) and very strongly penalizes unsuccessful ones, (e.g. hitting other traffic participants), can be used to train a reliable recognition and driving model [42]

7) And finally, real world driving experience should be used to update the model.
The reason I suggest to involve simulated environment is that it allows for generating a dataset with ground truth, which may open the possibility for a deep learning based approach that learns the mapping between the raw video, the lane, the environment and driver's response. Among other advantages is the possibility to train and evaluate the solution on very unusual and risky environments – experiences which are not always viable or even possible to obtain in real world.
The final outcome of the next thesis is to arrive to a point where algorithm is able learn to drive autonomously the routes taken by drivers frequently.

Bibliography

C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®, By: Kate Gregory and Ade Miller, Publisher: Microsoft Press

K. Kluge, "Extracting Road Curvature and Orientation from Image Edge Points without Perceptual Grouping into Features," in Proceedings Intelligent Vehicles Symposium, pp. 109-114, 1994.

B. Serge and B. Michel, "Road Segmentation and Obstacle Detection by a Fast Watershed Transform," in Proceedings of the Intelligent Vehicles '94 Symposium, pp 296-301, October 1994.

Y. Xuan, B. Serge and B. Michel, "Road Tracking, Lane Segentation and Obstacle Recognition by Mathematical Morphology," in Proceedings of the Intelligent Vehicles '92 Symposium, pp166-170, 1992.

K. Kluge and S. Lakshmanan, "A Deformable Template Approach to Lane Detection," in I. Masaky, editor, Proceedings IEEE Intelligent Vehicle'95, pp54-59, Detroit, September 25-26 1995.

S. Lakshmanan and K. Kluge, "Lane Detection for Automotive Sensor," in ICASSP, pp. 2955-2958, May 1995.

A. Broggi, "Robust Real-Time Lane and Road Detection in Critical Shadow Conditions," in Proceedings IEEE International Symposium on Computer Vision, Coral Gables, Florida, November 19-21 1995. IEEE Computer Society.

A. Broggi and S. Berte, "Vision-Based Road Detection in Automotive Systems: a Real-Time Expectation-Driven Approach," Journal of Artificial Intelligence Research, 3:325-348, December 1995.

A. Broggi, "A Massively Parallel Approach to Real-Time Vision-Based Road Markings Detection," in Masaky, I. (Ed.), Proceeding IEEE Intelligent Vehicles'95, pp.84-89, 1995.

M. Bertozzi and A. Broggi, "GOLD: a Parallel Real-Time Stereo Vision System for Generic Obstacle and Lane Detection," IEEE Trans. Image Processing, pp62-81, Jan 1998.

D. Grimmer and S. Lakshmanan, "A Deformable Template Approach to Detecting Straight Edges in Radar Images," IE EE Trans. Pattern Analysis and Machine Intelligence, vol.18, pp.438-443, 1996.

D. Jung Kang, J. Won Choi and In So Kweon, "Finding and Tracking Road Lanes using LineSnakes," in Proceedings of Conference on Intelligent Vehicle, pp. 189-194, 1996, Japan.

Axel KASKE, Didier WOLF and Rene HUSSON, "Lane Boundary Detection Using Statistical Criteria," in International Conference on Quality by Artificial Vision, QCAV'97, pp. 28-30, 1997, Le Creusot, France.

Axel KASKE, Rene HUSSON and Didier WOLF, "Chi-Square Fitting of Deformable Templates for Lane Boundary Detection," in IAR Annual Meeting'95, November 1995, Grenoble France.

Intelligent Vision & Video, accessed on January 20, 2016 from https://www.altera.com/solutions/technology/intelligent-vision-and-video/overview.tablet.html

Introduction to FPGA Technology: Top 5 Benefits, retrieved March 10, 2016 from http://www.ni.com/white-paper/6984/en/#

Cyclone V Device Overview accessed on January 10, 2016 from https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf

CUDA C Programming Guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

GeForce GTX 660 Ti overview, accessed on January 2, 2016 from http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-660ti

Direct GPU-FPGA Communication, Alexander Gillert, April 15, 2015

Probabilistic Robotics, by Sebastian Thrun, Dieter Fox, Wolfram Burgard, 1999-2000 http://people.ufpr.br/~danielsantos/ProbabilisticRobotics.pdf

Parallel Mandelbrot in Julia, C++, and OpenCL retrieved on April 2016 from http://distrustsimplicity.net/articles/mandelbrot-speed-comparison/

Computer Vision for the Web, by: Foat Akhmadeev retrieved on April 2016 rom http://proquest.tech.safaribooksonline.de.eaccess.ub.tum.de/book/programming/javascript/9781785886171/what-is-filtering-and-how-to-use-it/ch02lvl2sec13_html?query=((image+convolution))#snippet

OpenCL Basics: Flags for the creating memory objects, retrieved March 10, 2016, https://streamcomputing.eu/blog/2013-02-03/opencl-basics-flags-for-the-creating-memory-objects/

Creating and Managing Buffer Objects In OpenCL, retrieved March 15, 2016, from https://developer.apple.com/library/mac/documentation/Performance/Conceptual/OpenCL_MacProgGuide/CreatingandManagingBufferObjectsInOpenCL/CreatingandManagingBufferObjectsInOpenCL.html

The Khronos Group Inc., clCreateBuffer, retrieved October 15, 2015 from https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateBuffer.html

OpenCL Basics: Flags for the creating memory objects, retrieved October 25, 2015 from https://streamcomputing.eu/blog/2013-02-03/opencl-basics-flags-for-the-creating-memory-objects/

OpenCL Programming by Example, by: Ravishekhar Banger; Koushik Bhattacharyya
http://proquest.tech.safaribooksonline.de.eaccess.ub.tum.de/book/programming/9781849692342/3dot-opencl-buffer-objects/ch03s04_html?query=((clEnqueueReadBuffer))#snippet

The Khronos Group Inc., clEnqueueWriteBuffer, retrieved on January 2016 from
https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clEnqueueWriteBuffer.html

The Khronos Group Inc., clEnqueueReadBuffer retrieved on January 2016 from
https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clEnqueueReadBuffer.html

Heterogeneous Computing with OpenCL, by: David R. Kaeli; Perhaad Mistry; Dana Schaa; Dong Ping Zhang2.0, accessed on January 10, 2016 from
http://proquest.tech.safaribooksonline.de.eaccess.ub.tum.de/book/programming/9780128016497/chapter-3-introduction-to-opencl/s0060_html_3?query=((clSetKernelArg))#snippet

OpenCL Programming by Example, By: Ravishekhar Banger; Koushik Bhattacharyya
http://proquest.tech.safaribooksonline.de.eaccess.ub.tum.de/book/programming/9781849692342/5dot-opencl-program-and-kernel-objects/ch05s02_html?query=((clSetKernelArg))#snippet

The Khronos Group Inc., clSetKernelArg , retrieved on January 2016 from
https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clSetKernelArg.html

The Khronos Group Inc., clEnqueueNDRangeKernel retrieved on January 2016 from
https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clEnqueueNDRangeKernel.html

Threading Building Blocks (Intel® TBB), accessed on November 20, 2015 from
https://www.threadingbuildingblocks.org/intel-tbb-tutorial

Direct GPU-FPGA Communication, Alexander Gillert, April 15, 2015

The Benefits to Working in RGB from http://bigpicture.net/node/2391

Grapics Recognition, Algorithms and Applications, Dorothea Blostein, Young Bin Kwon, 2001, Kingston, Ontario

Real-time lane detection and tracking on high performance computing devices (Botsch, 2015).

Recent Progress in Road and Lane Detection – A survey Aharon Bar Hillel Ronen Lerner Dan Levi Guy Raz

Hardware Accelerated Particle Filter for Lane Detection and Tracking and OpenCL, Nikhil Madduri, 2014