Hardware Accelerated Particle Filter for Lane Detection and Tracking in OpenCL

Nikhil Madduri
2014

Master of Science (120 credits)
Space Engineering - Space Master

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Hardware Accelerated Particle Filter for Lane Detection and Tracking in OpenCL

Thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Space Science and Technology

on 28th Jan, 2014

Author
Nikhil Madduri

Advisor
Dr. Sergio Montenegro
Professor and Chairperson
Chair VIII, Department of Informatics
Julius-Maximilians-Universität Würzburg

Examiner
Dr. Jana Mendrok
Assistant Professor
Division of Space Technology
Luleå University of Technology

Supervisor
Sebastian Klose
Research Associate
Department of Informatics
Technische Universität München
Declaration of Authorship

I, Nikhil MADDURI, declare that this thesis titled, 'Hardware Accelerated Particle Filter for Lane Detection and Tracking in OpenCL.' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.

- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.

- Where I have consulted the published work of others, this is always clearly attributed.

- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

- I have acknowledged all main sources of help.

- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____________________________________________

Date: _______________________________________________
“There are no passengers on Spaceship Earth. We are all crew.”

Marshall McLuhan, 1964
Abstract

Faculty of Robotics and Telematics
Department of Computer Science

Master of Science in Space Science and Technology

Hardware Accelerated Particle Filter for Lane Detection and Tracking in OpenCL.

by Nikhil Madduri

A road lane detection and tracking algorithm is developed, especially tailored to run on high-performance heterogeneous hardware like GPUs and FPGAs in autonomous road vehicles. The algorithm was initially developed in C/C++ and was ported to OpenCL which supports computation on heterogeneous hardware. A novel road lane detection algorithm is proposed using random sampling of particles modeled as straight lines. Weights are assigned to these particles based on their location in the gradient image. To improve the computation efficiency of the lane detection algorithm, lane tracking is introduced in the form of a Particle Filter. Creation of the particles in lane detection step and prediction, measurement updates in lane tracking step are computed parallelly on GPU/FPGA using OpenCL code, while the rest of the code runs on a host CPU. The software was tested on two GPUs - NVIDIA GeForce GTX 660 Ti & NVIDIA GeForce GTX 285 and an FPGA - Altera Stratix-V, which gave a computational frame rate of up to 104 Hz, 79 Hz and 27 Hz respectively. The code was tested on video streams from five different datasets with different scenarios of varying lighting conditions on the road, strong shadows and the presence of light to moderate traffic and was found to be robust in all the situations for detecting a single lane.
Acknowledgements

I would like to thank Sebastian Klose (Department of Informatics, Technische Universität München (TUM)) for supervising my thesis, giving invaluable suggestions at every phase and for his incredible support. I would like to thank Dr. Sergio Montenegro (Department of Informatics, Julius-Maximilians-Universität Würzburg) for accepting my thesis proposal and being supportive of my work as the thesis advisor. I would like to sincerely thank Dr. Jana Mendrok (Luleå University of Technology, Sweden) for carefully going through the entire thesis and suggesting numerous modifications that only made my thesis more elegant. I would like to thank Dr. Kai Huang and Hardik Shah (Department of Informatics, TUM) for offering me this interesting thesis work in the first place.

I must thank the European Commission for granting me the Erasmus-Mundus scholarship without which this dream of pursuing Master’s would never have come true. I would like to thank Dr. Victoria Barabash (Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, Sweden) and the SpaceMaster consortium for organizing such a very useful program in the field of Space Science and Technology.

I thank my mom and dad for their incredible love and support without which I would not have been able to make it all the way from India to pursue Master’s in Europe.
Contents

Declaration of Authorship i

Abstract iii

Acknowledgements iv

Contents v

List of Figures vii

List of Tables x

Abbreviations xii

1 Introduction 14
  1.1 Motivation and Purpose ........................................ 14
     1.1.1 Problem Statement ........................................ 14
     1.1.2 Recommended Solution .................................... 15
     1.1.3 List of Contributions ..................................... 16
  1.2 The Structure of the Thesis .................................... 17

2 Background 18
  2.1 Lane Detection .................................................. 19
     2.1.1 Selection of the Region of Interest ..................... 19
     2.1.2 Correction for the Perspective Effect .................. 20
     2.1.3 Gray-Scaling, Thresholding and Edge Detection ......... 20
     2.1.4 Line Fitting ............................................... 21
  2.2 Lane Tracking .................................................. 25
     2.2.0.1 Particle Filter .......................................... 25
  2.3 High Performance Computing and OpenCL ..................... 26
     2.3.1 Programming Models for High Performance Computing ... 28
     2.3.2 Open Computing Language (OpenCL) ...................... 30
**Contents**

2.3.2.1 Conceptual Foundations of OpenCL .......... 30
2.3.2.2 Some Advanced Features in OpenCL .......... 37

3 Related Work ......................................... 40
3.1 Lane Detection ...................................... 40
3.2 Lane Tracking Methods .............................. 43
    3.2.1 Kalman Filter .................................. 43
    3.2.2 Particle Filter ................................. 45

4 Method and Implementation ......................... 47
4.1 Developing lane detection and tracking system for CPU ........ 47
    4.1.1 Preprocessing the Image ....................... 47
    4.1.2 Lane Detection ................................ 49
    4.1.3 Lane Tracking ................................ 52
        4.1.3.1 Particle Filter for Tracking a Robot in 2D .... 53
        4.1.3.2 Particle Filter for Tracking Road-Lane Markings . 56
4.2 Parallelizing the Algorithm for OpenCL ........... 58
    4.2.1 OpenCL Software Implementation ............... 58
    4.2.2 Hardware Architecture of OpenCL Devices Used .... 61
        4.2.2.1 GPU Specifications .......................... 61
        4.2.2.2 FPGA Specifications ......................... 63

5 Results .................................................. 65
5.1 Lane Detection ...................................... 65
5.2 Particle Filter for Tracking a Robot in 2D .......... 69
5.3 Particle Filter for Tracking Road-Lane Markings .... 71
    5.3.1 TUM Night Dataset ............................. 73
    5.3.2 TUM Day Dataset ............................... 78
    5.3.3 Caltech Cordova1 Dataset ..................... 78
    5.3.4 Caltech Cordova2 Dataset ..................... 80
    5.3.5 Caltech Washington Dataset .................... 80
5.4 Performance of the OpenCL Implementation on Heterogeneous Hardware ..................................... 81

6 Conclusion ............................................. 90

A Lane Markings at Intersections ....................... 92

Bibliography ............................................. 93
## List of Figures

2.1 Difference between least squares & total least squares methods. . . . 22  
2.2 Mapping from $x - y$ plane to $r - \theta$ plane. .............................. 23  
2.3 Lines passing through points in $x - y$ plane represented as curves in $r - \theta$ plane. ................................................................. 24  
2.4 Heterogeneous hardware. ............................................................... 30  
2.5 OpenCL Platform Model. ................................................................. 32  
2.6 OpenCL host and devices. ................................................................. 33  
2.7 NDRange Indices of work items. ....................................................... 33  
2.8 NDRange for 1D, 2D and 3D. ............................................................ 34  
2.9 OpenCL memory model. ................................................................. 35  
4.1 Lane Detection and Tracking Schematic ........................................... 48  
4.2 ROI selection. .................................................................................. 48  
4.3 Edge detection. ................................................................................. 49  
4.4 Lane Detection Schematic ................................................................. 50  
4.5 Line Sampling ................................................................................... 51  
4.6 Before introducing distance criteria. ................................................. 52  
4.7 After introducing distance criteria. .................................................... 52  
4.8 Lane Tracking Schematic ................................................................. 53  
4.9 The figures above depict how the particle filter is initialized with sample particles. ............................................................ 54  
  a Robot in an empty workspace of size 100x100. ................................. 54  
  b Robot in the workspace populated with particles. ............................ 54  
4.10 Good lines and the best lines. .......................................................... 57  
  a Original Image .................................................................................. 57  
  b Particle Filter in action. ................................................................. 57  
4.11 Lane Detection and Tracking using OpenCL ..................................... 59  
4.12 Schematic for OpenCL Execution on GPUs .................................... 60  
4.13 Schematic for OpenCL Execution on FPGA .................................... 61  
5.1 Test conditions given in Table 5.1 ..................................................... 66  
  a Original image. .................................................................................. 66  
  b After lane detection. ................................................................. 66  
5.2 Test conditions given in Table 5.1 ..................................................... 67  
  a Original image. .................................................................................. 67  
  b After lane detection. ................................................................. 67
5.3 Test conditions given in Table 5.1. ........................................... 67
a Original image. ................................................................. 67
b After lane detection. ......................................................... 67
5.4 Test conditions given in Table 5.1. ........................................... 68
a Original image. ................................................................. 68
b After lane detection. ......................................................... 68
5.5 Test conditions given in Table 5.1. ........................................... 69
a Original image. ................................................................. 69
b After lane detection. ......................................................... 69
5.6 Particles after re-sampling phase, for iteration 4. ......................... 70
5.7 Particle-weight distribution for iteration-4. ................................. 71
5.8 Robot and particles after measurement update. ............................. 72
5.9 ................................................................. 75
a Original Image. ................................................................. 75
b TUM Night - Best and Good Lines ........................................ 75
c After the lane tracking step shown in Figure 5.9b, post-
processing is done to introduce a horizontal search windows
(orange lines) at a few discrete points along the best lines
(red lines). Once a certain gradient is encountered along
these search windows, the pixels are colored blue. This step
increases the accuracy in locating the actual lane markings. 75
5.10 A few snapshots from the results for TUM Night Dataset. Figures
in the left column are the original images while in the right are
the lane-detected images. The blue colored markings are the lanes
as detected by our algorithm. .................................................. 76
5.13 Likelihood Vs Frame Number .................................................. 78
5.14 A few snapshots from the results for TUM Day Dataset. Figures
in the left column are the original images while in the right are the
lane-detected images. The blue colored markings are the lanes as
detected by our algorithm. .................................................. 79
5.15 A few snapshots from the results for Caltech Cordova1 Dataset.
Each original snapshot is followed by its corresponding lane-detected
image. The green lined borders in the lane-detected images indi-
cate the ROI while the red colored lines are the lane markings as
detected by our algorithm. .................................................. 80
5.16 A few snapshots from the results for Caltech Cordova2 Dataset.
Each original snapshot is followed by its corresponding lane-detected
image. The green lined borders in the lane-detected images indi-
cate the ROI while the red colored lines are the lane markings as
detected by our algorithm. .................................................. 81
5.17 A few snapshots from the results for Caltech Washington Dataset.
Each original snapshot is followed by its corresponding lane-detected
image. The green lined borders in the lane-detected images indi-
cate the ROI while the red colored lines are the lane markings as
detected by our algorithm. .................................................. 82
5.18 Average frame rate Vs Number of Particles. . . . . . . . . . . . . . 85
A.1 Horizontal Markings. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Key Hardware Specifications for GPUs Used</td>
<td>62</td>
</tr>
<tr>
<td>4.2</td>
<td>Key Hardware Specifications for CPUs Used</td>
<td>63</td>
</tr>
<tr>
<td>4.3</td>
<td>Key Hardware Specifications for the FPGA Used</td>
<td>64</td>
</tr>
<tr>
<td>5.1</td>
<td>Conditions used for various test cases of lane detection system.</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>General Test Conditions</td>
<td>74</td>
</tr>
<tr>
<td>5.3</td>
<td>Memory Size used by Kernel Arguments</td>
<td>84</td>
</tr>
<tr>
<td>5.4</td>
<td>Test Conditions</td>
<td>86</td>
</tr>
<tr>
<td>5.5</td>
<td>FPGA Resource Utilization for Executing Kernels for <strong>TUM Night Dataset</strong></td>
<td>88</td>
</tr>
<tr>
<td>5.6</td>
<td>FPGA Resource Utilization for Executing Kernels for <strong>TUM Night Dataset</strong></td>
<td>89</td>
</tr>
</tbody>
</table>
List of Algorithms

4.1 Particle Filter for Tracking a Robot .................................. 54
4.2 RESAMPLE ........................................................................... 56
4.3 Particle Filter for Lane Tracking ............................................ 57
Abbreviations

1D/2D/3D 1-Dimensional/2-Dimensional/3-Dimensional
ACDS Altera Complete Design Suite
API Application Programming Interface
BSP Bayesian Signal Processing
CPU Central Processing Unit
CUDA Compute Unified Device Architecture
DSM Distributed Shared Memory
DSP Digital Signal Processor
DAS Driver Assistance Systems
ECU Electronic Control Unit
EKF Extended Kalman Filter
FLOPS Floating Point Operations Per Second
FPGA Field Programmable Gate Array
GFLOPS Giga Floating Point Operations Per Second
GPGPU General Purpose Graphics Processing Unit
GPU Graphics Processing Unit
HHPC Heterogeneous High Performance Computing
HPC High Performance Computing
HT Hough Transform
IPM Inverse Perspective Mapping
LDS Lane Detection System
LDTS Lane Detection and Tracking System
LIDAR A portmanteau of LIGHT and RADAR
MCMC Markov Chain Monte Carlo
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>NDRange</td>
<td>N-Dimensional Range</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language</td>
</tr>
<tr>
<td>OpenCV</td>
<td>Open source Computer Vision library</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multi-Processing</td>
</tr>
<tr>
<td>PCIe</td>
<td>Peripheral Component Interconnect express</td>
</tr>
<tr>
<td>PDF</td>
<td>Probability Distribution Function</td>
</tr>
<tr>
<td>PGAS</td>
<td>Partitioned Global Address Space</td>
</tr>
<tr>
<td>RADAR</td>
<td>RAdio Detection And Ranging</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RANSAC</td>
<td>RANdom SAmple Consensus</td>
</tr>
<tr>
<td>RLD</td>
<td>Randomized Line Detection</td>
</tr>
<tr>
<td>RODT</td>
<td>Real Orientation Distance Transform</td>
</tr>
<tr>
<td>ROI</td>
<td>Region Of Interest</td>
</tr>
<tr>
<td>SHT</td>
<td>Statistical Hough Transform</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SM</td>
<td>Streaming Multi-processors</td>
</tr>
<tr>
<td>SPMD</td>
<td>Single Program Multiple Data</td>
</tr>
<tr>
<td>SP</td>
<td>Streaming Processor</td>
</tr>
<tr>
<td>TFLOPS</td>
<td>Tera FLoating Point Operations Per Second</td>
</tr>
<tr>
<td>TPC</td>
<td>Thread Processing Clusters</td>
</tr>
<tr>
<td>TUM</td>
<td>Technische Universität München</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
</tbody>
</table>
Dedicated to my parents.
Chapter 1

Introduction

1.1 Motivation and Purpose

1.1.1 Problem Statement

Road accidents lead to several millions of fatalities every year around the world (Muad et al. (2002), Aly (2008), Borkar et al. (2009b)). Studies show that a significant number of these are caused due to the driver distractions. Number of such accidents can be lowered by enabling Driver Assistance Systems (DAS). While DAS includes numerous technologies, Lane Detection is one such technology (Ruyi et al. (2011)) that has grabbed significant attention from mid-1980’s from researchers in this field. Lane Detection systems deal with detecting the lane markings on the road using mainly, image processing techniques on the images captured from a camera on board automobiles. This can help in developing warning systems to alert a driver of an impending accident. This is particularly a challenging problem due to varying light conditions, traffic on the road that obstructs the lane markings, the shadows cast by buildings or trees. While developing a robust algorithm for lane detection in itself is a complex task, there are limited computational resources and energy available in an automobile within which the DAS should work.

Traditionally, computations are performed on ECUs (Electronic Control Unit) in automobiles. However, one needs hundreds of ECUs in a car to meet all the requirements such as entertainment, information and safety critical systems (like DAS). And also, it takes kilometers long, complex wiring to get all these ECUs
to work together. More wiring also implies additional weight, which would also reduce the mileage of the vehicle per unit fuel burnt.

The Issues

The two major issues that are dealt with in the current work are:

1. Developing Lane Detection System (LDS) for DAS.

2. Suggesting alternate hardware solution that can replace traditional ECUs in future automobiles, by demonstrating the implementation of LDS (1) on heterogeneous, high-performance hardware such as GPUs and FPGAs.

1.1.2 Recommended Solution

In order to resolve the issues mentioned above, the approach we would like to take is as follows:

1. Develop a LDS on a standard CPU, using C/C++ programming language.

   In lane detection, the lane markings are detected in every frame without gathering any information from the previous frame computations. This increases the cost of computation and lowers the throughput of the algorithm. Hence, it can help to keep the information of the past computations so that the present computation doesn’t need a high computational effort. Lane Tracking is a method by which a line that is detected in one frame can be tracked in several consecutive frames. We use a Particle Filter for Lane Tracking.

2. Port the C/C++ code to Open Computing Language (OpenCL) that enables us to test LDS algorithm developed in (1), on high-performance, energy efficient, heterogeneous hardware.

   We carry out the testing on three different hardware:
   
   - GPU: NVIDIA GeForce GTX 660 Ti,
   - GPU: NVIDIA GeForce GTX 285 and
   - FPGA: Altera Stratix-V.
Choice of OpenCL  There are several other languages that can be used to pro-
gram GPUs and Multi-core CPUs (e.g., CUDA by NVIDIA, Knights Ferry by
Intel, DirectCompute by Microsoft) and FPGAs (e.g., VHDL) but all of them
are vendor-specific, which means they work only on the hardware they were in-
tended for. Hence porting them to a different hardware can involve significant
work. OpenCL on the other hand is the first industry standard for parallel com-
puting along with the support for heterogeneous hardware. Hence a code written
in OpenCL can be ported to GPUs, FPGAs, DSPs and Multi-Core CPUs alike,
without major modifications. This helps in developing software that runs on het-
erogeneous hardware in future automobiles, which can in turn help in replacing
the traditional ECUs. More information on OpenCL can be found in chapter 2.

Choice of Particle Filter  Particle Filter is chosen in the current work for
Lane Tracking. Other most popularly used method in Lane Tracking is Kalman
Filter. One of the major aspects of the current work being parallel computing on
heterogeneous hardware, the choice of algorithm should also support this. Kalman
Filter, while being very popular for the past few decades in almost every scientific
field, it is an estimator for linear dynamic system perturbed by Gaussian noise
(Grewal et al. (2001)). This means that it comes with certain assumptions that
the states to be extracted from a given signal are linear and the noise is Gaussian,
hence Kalman Filter is continuous but unimodal. On the other hand Particle Filter
is continuous, multi-modal, non-linear and is quite suitable for parallel processing
as each particle can be processed independent of the other particles. It is also
computationally much lighter as compared to Kalman Filter.

1.1.3 List of Contributions

Given below is the list of my contributions to the current work:

- Developing C/C++ code for Lane Detection and Tracking System (LDTS).
- Testing LDTS on several urban and sub-urban road scenarios in varying
  light conditions.
- Porting C/C++ based LDTS code to OpenCL.
• Testing the OpenCL based LDTS code on two GPUs and an FPGA mentioned before, and making performance comparisons between these hardware.

1.2 The Structure of the Thesis

This thesis is organized as follows - chapter 2 gives a background on some general concepts involved in Lane Detection that will be useful to understand terminology in the rest of the thesis. It also provides a quick introduction to OpenCL. chapter 3 presents a survey on the relevant literature in lane detection and lane tracking field. In chapter 4, a detailed explanation for the entire method and implementation of the current work is presented, while chapter 5 provides with all the results obtained in the current work and their explanation. Finally, chapter 6 ends the thesis with concluding remarks.
Chapter 2

Background

In this chapter, a brief conceptual summary of lane detection, lane tracking and OpenCL is presented. As mentioned in chapter 1, lane detection is a computationally expensive step. Hence, lane tracking is performed to speed up the algorithm. The code is finally ported to OpenCL so that it can be tested on GPUs and FPGAs. While this chapter mainly presents theoretical background, more details specific to our method and implementation are presented in chapter 4.

In section 2.1, major sub-blocks included in lane detection are presented. It involves selecting a Region of Interest (ROI) in which the lanes are to be detected, followed by removing the perspective effects in the image as the camera mounted on a car will usually have a perspective which needs to be corrected for. This is followed be gray-scaling, thresholding and edge-detection stages. Finally, line models need to be fitted to the detected edges to visualize the lane markings on the road.

In section 2.2, a brief description of the Particle Filter used for lane tracking is provided. Finally, in section 2.3, background of the programming models for High Performance Computing (HPC) is presented. An explanation for why OpenCL was chosen in the current work among so many parallel programming models is also presented.

While some of the concepts such as Hough Transform (section 2.1) and Inverse Perspective Mapping (subsection 2.1.2) presented in this chapter were not implemented in the current work, the background provided here would help in understanding the differences these methods have, with our implementation as described in the subsequent chapters.
2.1 Lane Detection

2.1.1 Selection of the Region of Interest

The Region Of Interest (ROI) is any rectangle that defines a sub-region of the original image, that should be considered by the lane detection system to perform computations so as to detect and extract the lane markings. Naturally, this has to be user-defined, as per the requirement.

It is a general practice to select a ROI as a first step to solving the lane detection problem. If carefully chosen, the ROI can enhance the speed and accuracy of detecting the lane markings on a given image (Borkar et al. (2009b)). This considerably reduces the number of image pixels on which the computation is to be carried out, hence improving the execution speed for the algorithm several times than if the whole image was considered. Moreover, it makes more sense to avoid processing the regions of the image, where there is a high chance that there will be no lines to be detected. For example, when a 2D image captured from a camera placed on a car’s dashboard is considered, the lower most region of the image may consist of the car’s hood, middle region - the road, the grass on either side of the road, the traffic and the top region - the sky, the trees and may be sign boards.

However, this may not always be the case as it depends on the camera placement, the height of the dashboard above the ground etc., hence, the algorithm for the automatic selection of the ROI has to be customized as per the requirement. Another important criterion that one might need to consider for selecting the ROI is the side of the road one has to follow while driving, based on the traffic rules because, while driving on a road in Germany, one has to be on the right side while in the United Kingdom, one has to be on the left side. If one drives on the right side of the road, the image captured might be filled with grass on the right side while the left side still has the road. Hence, it is a suitable choice to crop a little region on the right side in such an image. Similarly, it might be more appropriate to crop the image on the left if the image is captured while driving on the left side of the road.

Other factors that should be considered while deciding the ROI are the lighting conditions while capturing the image. If it is night time, the light beam from the vehicles on the opposite side of the road can make the road too glossy to detect
the lines on the road. Hence, a part of the image may be difficult to process, thus making it logical to avoid such regions from the ROI. Similarly, the road signs on either side of the road, which glow with high intensity when the vehicles’ head light beam hits them can be carefully avoided by choosing the appropriate ROI.

Considering all the above criteria, and adding some more suited for a given scenario, one might end up having the ROI as small as only 15-40% in height of the original image along the vertical axis and 50-80% in width, along the horizontal axis. In Aly (2008), it was mentioned that an ROI of $160 \times 120$ pixels was chosen from an original image of $640 \times 480$ pixels which means a ROI of 40% in height and 40% in width was chosen from the original image.

### 2.1.2 Correction for the Perspective Effect

When a 2D image is captured by a camera, there are optical distortions because the real world is in 3D and the camera is transforming and representing it in 2D. From the same explanation, arises the term Vanishing Point in the road scenario. The appearance of the road as viewed in a 2D image gives a perception as if the road’s edges converge to a point at a distance ahead, while in reality, they are parallel and do not intersect. The point where they appear to meet is called the *Vanishing Point*. Since a 2D image lacks the information on the third dimension, it is a good practice to reconstruct the top-view (bird’s eye view) of the road using various methods for transformation to obtain the image on which the road boundaries no more intersect with each other at the Vanishing Point but are parallel to each other. The top view makes it easier to detect the lanes on the road. One of the methods commonly used to obtain the top-view is Inverse Perspective Mapping (IPM) (Lai et al. (2000), Muad et al. (2002), Bertozzi et al. (1996) and Aly (2008)).

### 2.1.3 Gray-Scaling, Thresholding and Edge Detection

After removing the perspective effect in the image by applying IPM technique, the image is converted into gray-scaled image, often followed by thresholding. Thresholding is a technique by which a gray-scaled image retains only the pixels that have gray-scale intensity higher than a threshold value. All the pixels with less intensity will be assigned a zero value. To detect the edges in the image that would give clues regarding where the lane markings can be, a Sobel filter (Pradabpet et al. (2009), Ma et al. (2010b)) or a Canny edge detector (Canny (1986)) is generally
used. In the current work, we have used Sobel detector program provided by
the *Open source Computer Vision library* (OpenCV) that can be included in the
C/C++ program.

Once the edges are detected, to visualize the high-gradient edges, line or
curve models need to be fitted to these edges. Procedure described in the next
section describes the most commonly used methods.

### 2.1.4 Line Fitting

**Method of Least Squares** In case of lane detection, since the simplest model
that can be used is line-fitting, consider Equation 2.1 representing a simple line
using *slope-intercept form*.

\[
y = mx + c \tag{2.1}
\]

Entire image is populated with such imaginary lines, with a chosen resolution
as per the speed and accuracy requirements of the least squares approach. For
every pixel in the image with *x-coordinate* of \(x_i\), the aim is to find that line which
best predicts \(y_i\). However, it is not practical to populate the image with very
high resolution of imaginary lines such that for every \(x_i\), there is always a line
which exactly satisfies the Equation 2.1. Hence, an error is associated to the line,
each time a pixel \((x_i, y_i)\) is approximated to be lying on the nearest line a certain
distance along the *y-axis*. In Figure 2.1, the sub-figure on the left depicts this
method. The essence of the *least squares* method lies in minimizing this error so
that an edge is approximates by fitting the line that is close to more number of
points, by summing squares of the errors across all the data points \((x_i, y_i)\) in the
image, as given by

\[
\sum_i (y_i - mx_i - c)^2. \tag{2.2}
\]

However, a major flaw in representing the lines by Equation 2.1 is that it
cannot represent the vertical lines. Hence, a much better representation to a
line would be the *standard form* given in Equation 2.3, which does not have the
shortcomings of the *slope-intercept form*.

\[
a x + by + c = 0 \tag{2.3}
\]
With this representation, the aim is to minimize

$$\sum_i (ax_i + by_i + c)^2.$$  \hspace{1cm} (2.4)

This more accurate method is known as the method of total least squares. In Figure 2.1, the sub-figure on the right shows the error-measuring scheme of the total least squares methods.

**Figure 2.1:** This figure from Ponce et al. (2012) shows the difference in the error-measuring scheme of least squares (left) and the total least squares (right) methods.

**Hough Transform** Hough Transform (HT) is a quite generic model to fit a known structure to a set of tokens. These tokens can be the pixels along a desired shape in a given image. However, as the complexity of the shape increases, the computation effort (discussed later in this section) increases exponentially for HT.

A voting procedure is carried out to vote the structure on the basis of the number of tokens that the structure passes through. Hence, the higher the number of votes that a structure gains, the better it represents the feature in the image that the structure is to be fitted to. In the original method proposed by Hough (1962), the problem of detecting the lines was posed as finding the collinear points by using a slope-intercept parameter representation of straight-lines as in the Equation 2.1. However, there is a disadvantage of this parameterization because it is highly sensitive to the coordinate system chosen for the analysis of an image. All the lines which are very close to vertical have infinitely large slopes. Later, Rosenfeld (1969) has suggested a better approach which is independent of the chosen coordinate system i.e., finding the concurrent lines in the parameter plane instead of finding collinear points in the image plane. But even this approach has limitations because the slope and intercept are unbounded.
In Duda et al. (1972), a more usable parameterization has been suggested which is known as normal parameterization. This involves representing a straight line in the image plane as a single point in the parameter plane. Consider Figure 2.2 from Moeslund, T. (2009) in which each line in the $x - y$ plane maps to a point in $r - \theta$ plane.

![Diagram](image)

**Figure 2.2:** This figure from Moeslund, T. (2009) depicts the mapping of a line in $x - y$ plane to a point in $r - \theta$ plane, as generally used in Hough Transform for line-fitting.

In this figure, instead of the slope-intercept form (left) in, a line can be represented using the angle $\theta$ and the perpendicular distance, $\rho$ from the origin (right). Hence the equation for representing a line in the parameter plane becomes

$$x \cos(\theta) + y \sin(\theta) = \rho$$

(2.5)

for a given point $(x = x_0, y = y_0)$. However, if the values of $\theta$ are varied in the range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, in $n$-steps, there will be $n$-points in $r - \theta$ plane, all together representing a sinusoidal curve. Consider Figure 2.3 from Moeslund, T. (2009), which depicts this transformation from $x - y$ plane to $r - \theta$ plane for two points. Each of the two curves in the sub-figure on the right of Figure 2.3 represents all the lines that pass through the corresponding point on the sub-figure on the left.

The number of lines passing through a point depends on the choice of $n$ used to divide the range of $\theta$. As can be seen, the two curves on the right of Figure 2.3 intersect. As each point in $r - \theta$ plane represents a line in $x - y$ plane, the point of intersection of two curves in $r - \theta$ plane would then represent the line on which the two points in the $x - y$ plane lie. Hence, if we perform a similar transformation for every point in $x - y$ plane, we have as many curves in $r - \theta$ plane as there are point in $x - y$ plane. Each of such curves intersects at various points at several
other curves. According to Duda et al. (1972), $r - \theta$ space is divided into finite number of grid points along each axis of the parameter space and the array thus obtained is called an accumulator array. A counter is assigned and initialized to '0' at each grid point. Corresponding counter is incremented by 1, every time a set of curves intersect at a particular grid point. Using the usual consensus that an edge in an image is made up of a large number of collinear or nearly collinear points, the aim is then to chose those grid points in the array where maximum number of curves have intersected. Hence, the original problem of finding collinear points, as suggested by Hough (1962) is reduced to finding intersecting curves in the parameter space.

Hough Transform suffers from its own problems like grid dimension, quantization errors and noise (Ponce et al. (2012)), to name a few. In case of simple straight line-fitting problems, the dimension of accumulator array is 2 because, only $r$ and $\theta$ are required for parameterization. If the problem is to use HT to fit circle, one needs a 3D parameter space as, the center of the circle and its radius have to be specified for a circle. Hence, as the structure that needs to be fitted using a HT increases in complexity, the computation time increases exponentially, thus making it impractical to use HT.

**Our Method** For fitting the lines to the edge detected image, we use a novel approach based on random sampling of the particles that characterize lines in the image. After populating the image with such line samples, weights are assigned to each line based on the sum of gradients in the neighborhood of such lines.
The higher the weight of a lane, the more is its chance to be representing the actual lane marking on the road. Further details on our method are presented in subsection 4.1.2.

2.2 Lane Tracking

Detecting the lines using lane detection techniques though has better accuracy, can be computationally very expensive as it requires populating the ROI with thousands of line samples and evaluating the weights of all of them. Repeating this step for every frame of the video stream can reduce the computation efficiency of the lane detection system using the limited computational hardware resources available on the automobiles and it is often difficult to meet the real-time requirements.

Hence, using the lane detection algorithm only for initialization and there onwards, keeping track of a few (e.g., only about 1/10\(^{th}\) of the number used for initialization) sample lines from the initialization can significantly improve the throughput of the algorithm. While other methods (e.g., Kalman Filter) can be used for this purpose, a Particle Filter can be more suitable when the form of the Posterior Probability Distribution Function (PDF) is unknown beforehand.

2.2.0.1 Particle Filter

Particle filter is a stochastic computational technique based on Bayesian Signal Processing (BSP) and Markov Chain Monte Carlo (MCMC) simulation models.

According to Candy (2009), Bayesian Signal Processing is concerned with the estimation of the underlying probability distribution of a random signal in order to perform statistical inferences. Basis for Bayesian techniques is the Bayes Rule given by

\[
Pr(X_i|Y_i) = \frac{Pr(Y_i|X_i) \times Pr(X_i)}{Pr(Y_i)} \tag{2.6}
\]

where \(X_i\) is the desired state variable that is inferred from the measurement \(Y_i\); the term \(Pr(X_i|Y_i)\) is called posterior probability distribution; \(Pr(Y_i|X_i)\) is likelihood, \(Pr(X_i)\) is prior probability distribution and \(Pr(Y_i)\) is the evidence, all at a given time \(t\). The notation \(Pr(A|B)\) denotes a conditional probability - the probability of an event 'A' occurring subject to the condition that event 'B' has already occurred.
Particle filter used in this work further follows a Markov Chain model because the desired state at any time depends only on the previous state but not the ones before. Particles are sampled randomly and do not follow any analytical distribution function and hence, particle filter also draws ideas from the Monte-Carlo simulation methods. A more detailed description of the implementation of Particle Filter is presented in subsection 4.1.3.

2.3 High Performance Computing and OpenCL

For over two decades until 2003, improvement in High Performance Computing (HPC) was primarily led by the microprocessor manufacturing companies constantly trying to improve the clock frequencies at which a single-core processor operates. However, the manufacturers and researchers in this field soon realized that it is not practically feasible to improve the single-thread performance way too beyond certain limits due to the issues with heat dissipation. This led to a new era in HPC, where multi-core CPUs started replacing the single-core CPUs. The energy consumed per clock cycle in switching the gates in a CPU can be expressed as

\[ Energy = Capacitance \times (Voltage)^2 \]  

(2.7)

If the CPU operates at frequency \( f \) clock cycles per second, the power consumption would then become \( Energy \times f \). Using these equations (Munshi et al. (2012)), one can find out that the power consumption of a single-core CPU operating at frequency \( f \) would consume about 2.5 times that consumed by a dual-core CPU with each core operating at a frequency of \( f/2 \) while both CPUs deliver the same throughput in terms of the number of operations performed in a given cycle. Hence, the HPC hardware has evolved to accommodate more and more cores on the same chip rather than increasing the clock speeds of a single-core.

Most of the current day CPUs come with multi-core microprocessors. However, to derive high-performance in scientific applications, the need of the day is to have many cores rather than simply, multiple cores. This lead to the GPUs, originally developed for fixed-function, graphics pipeline, to join the HPC. A decade ago, GPUs were meant for graphics rendering to perform operations of a standard graphics pipeline with the sequence - vertex operations, primitive assembly,
rasterization, fragment operations and composition (Owens et al. (2008)). Graphics pipeline, being hardware intensive and its steps vertex operations and fragment operations being highly parallelizable, required a microprocessor with many cores that can perform multiple operations in parallel and independent of each other. The GPUs that were in the consumer market prior to 2003 were non-programmable, hence rendering them inaccessible to the software developers to use the highly parallel GPU hardware for applications other than the graphics pipeline.

However, the trend has changed when one of the leading GPU manufacturers, NVIDIA had released its first programmable GPU, \textit{GeForce 3} in 2003, which was followed by the other manufacturers such as ATI with its Radeon 8500 and Microsoft with X-Box. This new class of GPUs started to be called GPGPUs (General Purpose GPUs) whose application is no more limited to the fixed-function, graphics pipeline but researchers in several other scientific domains like Computational Fluid Dynamics, Solar Physics, Bio-Mechanics or just about any field that had massive computational requirements started resorting to GPGPUs as they provided energy efficient and more importantly, low-cost solution to the problems that were previously computed using \textit{Supercomputers}.

One major challenge faced by the software programmers in the beginning phases of GPUs' migration to general purpose applications was that, though they became programmable, the software developers had to re-structure their code to suit the framework of the standard graphics APIs in order to tap the potential of GPUs in a completely alien domain that has nothing in common with graphics pipeline. Hence, the developers were still restricted to using GPUs only in certain applications. Even in the applications that suited the pipeline, it was a difficult and time consuming process to re-structure the code.

With the advent of new programming paradigms specifically meant for GPU programming, like NVIDIA’s 'C' like parallel programming model, CUDA that is supported by NVIDIA’s GPUs and OpenCL framework that is originally developed by Apple and now maintained by several major hardware and software companies headed by non-profit industry consortium \textit{Khronos Group}, GPGPU programming has never been more accessible to software developers. Moreover, unlike CUDA that is a proprietary framework which works only in NVIDIA GPUs, OpenCL supports portability across the multi-core CPUs, GPUs, DSPs and FPGAs from
several vendors. Hence OpenCL has opened new doors to Heterogeneous High Performance Computing (HHPC) using a single programming framework.

2.3.1 Programming Models for High Performance Computing

While there are a vast number of scientific fields constantly progressing to solve more complex problems everyday, there is one commonality in many of them - solving the problems faster than before, by using better computing hardware and software models. Hence, High Performance Computing (HPC) has pervaded into many of the fields in the current day scenario. While a decade ago, HPC was mainly performed using expensive supercomputing facilities, accessible only to well funded research organizations, today, HPC is something anyone can avail, using several heterogeneous low-cost hardware that is available in the market such as a combination of a multi-core CPU and a GPU, with comparable performances to the old supercomputers. One can use the GPUs available today to get a raw computing power of up to 1 TFLOPS (Tera FLOPS). Before leaping forward to study the features of OpenCL framework, it behooves to briefly discuss the evolution of HPC programming models over the past few decades. Several parallel programming models that have driven the HPC in the past as well as in the present are as follows:

- Pure parallel programming models
- Partitioned Global Address Space (PGAS) models
- Heterogeneous parallel programming models
- Hybrid Programming models

Traditionally programming models can be largely classified into two types - shared memory models and distributed memory models. While in the case of shared memory models (e.g. OpenMP, Pthreads), a single memory block is shared by several computing cores, in case of distributed memory model, each core has a separated memory block while messages are passed among several threads using programming models like MPI (Message Passing Interface). These memory models are usually called pure parallel programming models.
While shared memory models do not exploit the cache data locality, distributed memory models do, hence achieving better performance. Partitioned Global Address Space (PGAS) model addresses this issue by combining the shared and distributed memory models into what is known as Distributed Share Memory (DSM) model. However, PGAS model is not suitable for heterogeneous environments. While there are several programming models that support PGAS, like Unified Parallel C, Coarray FORTRAN, these language had to be created keeping exclusively PGAS model in view. However, since there already exist several programming models that support either shared or distributed memory models, there was a new trend of programming models that aimed at combining various models into Hybrid Programming models to extract the best in both shared and distributed memory architectures. Examples of such models are Pthreads (shared) combined with MPI (distributed), OpenMP (shared) combined with MPI (distributed), CUDA (shared) combined with MPI (distributed). Since OpenCL already has heterogeneous hardware support, it alone serves the purpose of any Hybrid models mentioned here, hence making OpenCL, a potential emerging Heterogeneous parallel programming paradigm in the future as the code written in OpenCL is not limited to particular platforms, but it is portable across multivendor hardware. There are several other programming frameworks (Diaz et al. (2012)) like NVIDIA’s CUDA, Knights Ferry by Intel, DirectCompute from Microsoft that can be used for heterogeneous hardware such as a combination of CPU and GPU. But most of these are vendor specific, hence rendering them unusable for heterogeneous hardware.

Figure 2.4 shows various computing hardware and their ability to support high-performance computing. While most of the modern computers come with multi-core CPUs, which to a lower degree support parallel computing, GPUs and FPGAs are used for many scientific applications to speed up the computation.

With more heterogeneity of available hardware, comes a difficulty in making the code developed for a particular platform, compatible with the other platforms which necessarily makes it impractical for a software developer to write a piece of code that runs on multiple devices. Hence, there is an increasing need to have an industry standard for programming heterogeneous devices, while providing a high-level abstraction for creating contexts, choosing devices, assigning tasks to various threads for parallel computing and load balancing etc, without the developer being overwhelmed with the low-level details of the hardware architecture. While there are several high-performance computing frameworks (software) from
several companies like NVIDIA, IBM, Intel & AMD, that support specific kind of hardware (usually belonging to the same company that created the software), an open standard, heterogeneous hardware support is a great flexibility to have for a software developer as the code developed for one platform can be ported, tested and used on an entirely different hardware, using the same basic code implementation. OpenCL (Open Computing Language) is one such industry standard that provides heterogeneous programming framework, managed by a non-profit technology consortium, Khronos Group (Gaster et al. (2011)). For example, one can port the code written and tested on an NVIDIA GPU to an Altera FPGA using OpenCL.

### 2.3.2 Open Computing Language (OpenCL)

#### 2.3.2.1 Conceptual Foundations of OpenCL

Open Computing Language (OpenCL) is the first open source, industry standard for parallel programming with a support for heterogeneous hardware such as multicore CPUs, GPUs, DSPs and FPGAs. Several documents can be accessed online from Khronos group’s webpage (OpenCL) and there exist some elaborate textbooks (Munshi et al. 2012, Gaster et al. 2011, Scarpino 2011) that explain a vast number of functionalities offered by OpenCL in detail. However, this subsection is meant for a brief mention of the underlying theoretical concepts and features that are provided as part of OpenCL framework.

Portability of the code being the major feature that sets apart OpenCL from other parallel programming models, for the same code to be interpreted the same
on heterogeneous devices, each device needs to have its own implementation of OpenCL. For example, OpenCL implementation provided by Altera for its FPGAs can vary from the implementation provided by NVIDIA for its GPUs. Hence, if there exist an Altera FPGA and an NVIDIA GPU in the same OpenCL platform, one needs to install both the implementations. Hence OpenCL implementation is usually provided by the vendors of the respective devices.

OpenCL divides the entire hardware available for computation into two parts - a host and the devices. Usually a host is a CPU and devices are GPUs, FPGAs or another set of multi-core CPUs. Using a set of APIs provided by OpenCL, one can develop a host code (usually C/C++) that identifies all the devices that are present in the platform, creates a context, enqueues commands and submits the device code to these devices. To understand how this works, one has to study the following models into which OpenCL divides any given problem:

1. Platform Model
2. Execution Model
3. Memory Model
4. Programming Model

**OpenCL Platform Model**

OpenCL abstracts the combination of host and devices as a single platform. Any platform has only one host (CPU) while devices (multi-core CPUs/GPUs/DSPs/FPGAs) can be one or many. Figure 2.6 shows an example of a CPU (OpenCL host) connected to GPUs and multi-core CPUs (OpenCL devices), where all the hardware and software put together be called an OpenCL Platform.

Any device under the OpenCL platform is termed as a compute device and is considered to be made of blocks of several compute units, while each compute unit is made up of several processing elements. Figure 2.5 presents the OpenCL’s Platform Model.

According to this model, a processing element is the smallest entity and it resembles a thread in traditional parallel programming terminology. This model is simply the abstraction used by OpenCL and has nothing to do with the actual hardware architecture itself. Hence this model is simply how OpenCL ”sees”
Chapter 2. Background

Figure 2.5: This figure from Munshi et al. (2012) depicts OpenCL’s Platform Model.

the devices under OpenCL platforms. OpenCL package when installed for any particular device contains broadly the following three items:

**OpenCL Execution Model**

To be able to execute the code on compute devices in parallel using OpenCL, one needs two programs:

1. Host side code: the code that chooses all the devices and assigns them tasks, sends data on which tasks are to be performed.

2. Device side code: these are called **Kernels**, the actual code that is meant to act on several threads at a time in parallel on OpenCL devices.

A host code is usually a C/C++ code. A kernel is a simple function that is written in OpenCL C language which is based on C99 and is contained in a file (additional file that can be called from within host code) with ".cl" as its extension (several other formats/extensions are possible). This file is the core to the entire idea behind OpenCL parallel programming. During the runtime, multiple instances of this same file are deployed in parallel on several processing elements (or threads) which act on multiple elements of a given data. **Figure 2.6** depicts where each of the **host-code** and the **device-code (kernel code)** is executed.

The OpenCL runtime environment assigns an index called **globalID** to each of the threads on which a kernel is deployed. These indices can vary from 1D to 3D arrays and hence, the range of such indices is generally termed as **NDRange**. Each globalID represents a **work item** in OpenCL’s terminology.
Figure 2.6: This figure from Alvarez et al. (2011) depicts how OpenCL’s host is connected to several compute devices and the respective locations where host-code and device-code (kernel code) are executed.

Work items are then organized into several blocks known as work groups and each work item within a work group is assigned another index called a localID. Hence, each work item is assigned two indices, one - the globalID to identify the work group globally and two - the localID to identify the work item within a work group. Figure 2.7 illustrates how the global and local ID’s are assigned in case of 2D range space of work items. Indices $(W_x, W_y)$ in the figure represent the number of work groups along a particular direction, $(G_x, G_y)$, the maximum globalID and $(L_x, L_y)$, the maximum localID possible along each direction.

Figure 2.7: This figure from Munshi et al. (2012) depicts how various indices are assigned to work items in a specific example of 2D work groups.
Figure 2.8 shows how the concept of NDRange for execution of work-items arranged into 1D, 2D and 3D work-groups might look. The work groups’ dimensionality and their sizes can affect the performance of the algorithm significantly. Hence, the algorithm designer must be decide which configuration of NDRange delivers the best performance.

![Figure 2.8: This figure from Alvarez et al. (2011) depicts how NDRange and the corresponding work-items & work-groups are arranged for 1D, 2D and 3D cases.](image)

OpenCL ensures that all the work items of a given work group are executed concurrently on processing elements of a single compute unit. However, one drawback of OpenCL 1.1 is that it cannot assure concurrency between the work items across different work groups (Munshi et al. (2012)).

To be able to execute an OpenCL program, a structure called **context** is created by calling certain OpenCL APIs. A context represents all the devices, kernels, **program objects** and **memory objects**. Program objects are built by OpenCL using the kernel files. Memory objects are all input and output data required to perform the parallel computation. Another structure called **command-queue** is created which maintains the order of execution of the commands specified within the host code. These commands can include **kernel execution commands**, **memory commands** and **synchronization commands**. Kernel execution commands are used to assign tasks to the processing elements and memory commands are used to transfer memory objects between host and the device. Synchronization commands can be used by the programmer to manually specify the points of synchronization where required within the program.
OpenCL Memory Model

OpenCL 1.1 supports two types of memory objects - **buffer objects** and **image objects**. Buffer objects are 1D arrays of data used by the kernels while performing computation. Image objects are meant for supporting images. However, in this work, we mostly use buffer objects.

In the host code, two kinds of memory objects are created - input memory objects and output memory objects which are either read (input) from or written (result/output) to from the kernels.

OpenCL divides the memory into various subregions as shown in the Figure 2.9. Contents placed in Host memory are accessible only to the host. Global memory gives read/write access to all the work items used in the program. Constant memory region is faster than global memory because though this is placed in the global memory, during the execution, the data specified as constant memory is copied onto the on-chip data cache blocks which can then be accessed faster by the work items. Local memory can be accessed by all the work items within a work group while Private memory, as the name suggests is accessible only to each work item. Usually private memory is implemented as registers in the device. OpenCL provides memory attributes named __global, __constant, __local and __private that can be attached to variable definition to control where each parameter/buffer resides in the memory.

![Diagram](image.png)

**Figure 2.9:** This figure from Chu et al. (2010) depicts the OpenCL memory model.
For better performance of a device, one has to understand and control the location of data into these regions of the OpenCL memory model as they can significantly effect the computation time. While global memory is the largest region available, storing the data in it can cause huge delays as each time a work item accesses a particular variable from global memory, the data has to be transferred via PCI express which is not a very fast mode. On the other hand, private memory is the fastest memory but it is very limited resource on any device. Hence small parameters like constant integers or floats can be stored in private memory. Local memory is also a limited resource as compared to the global memory but quite larger than private memory. It is usually made up of on-chip memory blocks. Hence, whenever possible, it is recommended to place the data buffers in this region instead of global memory.

Alvarez et al. (2011) presents a method on how to re-arrange the parallel program (kernel) to split the problem of simple \textit{matrix multiplication} into sub-problems so that only the data required by each sub-problem can be placed at any time in the local memory if the buffer for complete problem does not fit into this region.

\section*{OpenCL Program Model}

OpenCL mainly supports two kinds of parallel programming models - \textit{Data-Parallel} and \textit{Task-Parallel} models. Data-Parallel model involves a single program (kernel) operating on multiple elements of a data structure concurrently. Task-Parallel model can be defined as a set of independent programs operating concurrently.

In case of Data-Parallel model, though several data elements might use the same program (kernel), if there is conditional branching used in the program, the execution paths they follow can vary significantly. Such Data-Parallel models can further be classified as SPMD (Single Program Multiple Data) models while the models in which all the work-items follow identical path are called SIMD (Single Instruction Multiple Data) models. OpenCL supports both kinds of parallel programming models.

A combination of such models is also possible in OpenCL, for example, several concurrent tasks acting parallel on data. However, there are certain algorithms that cannot simply be implemented in OpenCL. Hence portability of OpenCL
across heterogeneous platforms comes with the cost of generality of the algorithms (Munshi et al. (2012)).

Contents of OpenCL Framework

When OpenCL 1.1 framework is installed on a machine, it consists of the following three essential components:

- OpenCL Platform APIs
- OpenCL Runtime APIs
- OpenCL Programming Language

While the first two items would be required to write a host side code, one has to write the device side code (kernels) in OpenCL programming language which is based on C99, hence it is basically a "C like" language.

2.3.2.2 Some Advanced Features in OpenCL

Since one of the important aspects of using OpenCL is high performance, it is required to analyze the time taken by individual blocks of the algorithm in addition to just measuring the total execution time. One can set flags at the start and end of all the essential blocks of the code and print out manually, the time taken for executing each block using for example, the API function clGetEventProfilingInfo as suggested by Mistry et al. (2011). This paper performs a thorough analysis of time profiling in OpenCL programs that use multiple kernels. Alvarez et al. (2011) suggests several useful tools provided by the device vendors, such as GPU visual profiler as part of ATI Stream SDK and OpenCL visual profiler that comes with CUDA toolkit by NVIDIA which can be used to visualize the resource utilization while the program executes and the algorithms’ performance.

As described in the section 2.3.2.1, the choice of work-group size significantly affects the overall performance of the OpenCL code. Chu et al. (2010) suggests a work-group size in the multiples of 64 to be optimal for ATI manufactured GPUs whereas a size of 32 gives the best performance in NVIDIA GPUs as NVIDIA’s CUDA architecture groups 32 threads into a single block which represents a work-group. Hence, the choice of an optimal work-group size not only depends on the
algorithm, but also the architecture of the compute device on which the computation is performed. This paper further shows that the GPUs often outperform even the expensive servers.

The execution time varies significantly based on the location where the memory objects are placed. If the read/write memory objects are placed in the global memory, the transfer of such memory between the host and devices via PCI express can be a very time-consuming step. Further, if there are several iterations that the code runs for and input for a new iteration depends on the output of the previous one, the costly memory transfer might have to be performed in every iteration which decreases the overall performance of the algorithm. However, OpenCL provides some advanced memory transfer mechanisms by which these host-device memory block transfers can either be completely avoided or reduced. One of these techniques includes using the API function `clEnqueueCopyBuffer` where the output buffer from the device is copied to another buffer on the device side without having to transfer to data all the way to the host and read it back for the next iteration’s input. Another technique called *Swap Method* is suggested by Alvarez et al. (2011) which involves simply swapping the buffers using the API function `clSetKernelArg` just before kernel execution. These techniques are most efficient because the bandwidths of the buses within a device are often up to 10 times or even faster than the PCI express over which the host and the device communicate.

**A Note on OpenCL for FPGAs & Embedded Systems**

GPUs and CPUs can be programmed using high-level languages like CUDA and C/C++. On the other hand, developing a hardware design for Field-Programmable Gate Arrays (FPGA) takes quite longer and considerable effort. There exist numerous *High-Level Synthesis (HLS)* tools like Vivado (AutoESL), Impulse C, MyHDL, FPGAC that use code written in high-level languages such as C/C++ & Python. But the sequential nature of such high-level languages do not allow for well-optimized hardware design suited for FPGAs.

Whether the FPGA designer uses low-level languages like VHDL, Verilog HDL or HLS tools, there is always a compromise either in terms of time and effort for designing the hardware or penalty of poor performance. OpenCL answers both of these issues. One can have low-level control over the hardware design using the tools such as Altera Complete Design Suite (ACDS) (Czajkowski et al. (2012)) or Xilinx AutoESL (Shagrithaya et al. (2013)) for generating the hardware design.
specification from OpenCL kernels, while the time and effort involved in designing OpenCL kernels can be appreciably lower.

OpenCL 1.1 further provides specification for embedded applications. Though embedded profile of OpenCL is always a strict subset of the complete set of APIs provided for the OpenCL’s full profile, this option is good to have as it allows for designing efficient parallel programs that run on FPGAs used in embedded applications.
Chapter 3

Related Work

3.1 Lane Detection

Significant research has already been done in developing lane detection systems for the past two decades. There are several sensors which are used for lane detection like camera/vision-based sensors, Line Sensors, LIDARs (Light RADAR) and GPS to name a few. Vision based sensors are usually more robust that other sensors in terms of its usability on variable road conditions and types of lane markings. However, a fusion of data from multiple types of sensors is a good way to go to get more accurate results. But such a method is out of scope of the current work and hence we limit ourselves to studying some of the existing literature using vision based lane detection techniques. Most of the vision based lane detection systems in follow quite a similar work-flow:

1. Capturing the image from the camera,
2. Cropping the image to obtain appropriate Region Of Interest (ROI),
3. Detecting the lines (the actual lane detection process) and finally,
4. Fitting straight line or curve models to the detected lane markings.

However, the actual process mentioned in item (3) above for detecting the lines varies significantly across several works published in the literature.
**Hough Transform**  One of the most commonly used methods for lane detection is Hough Transform (HT) (Hough (1962), Ballard (1987), Illingworth et al. (1988), Duda et al. (1972)). A brief description of HT is presented in subsection 2.1.4.

Aly (2008) uses an Inverse Perspective Mapping (IPM) to obtain a bird’s eye view of the image. IPM removes the perspective effect which makes the lane markings appear to be non-parallel when captured from the camera mounted on the car. After applying IPM, he uses HT to detect the lane markings which is followed by a Random Sample Consensus (RANSAC) line and curve fitting step. HT needs to maintain an accumulator array to store the information of the number of points in the image that represent each line, which means it consumes large storage. Especially when dealing with parallel computing, the memory from GPU/FPGA must be transferred to the host and vice-versa via PCIe card which is a slow process. Hence, this would inevitably slow down the efficiency of their method. HT suffers from its own problems like grid dimension, quantization errors and noise (Ponce et al. (2012)), to name a few. Sometimes the artifacts like cracks and tar patches might also be detected as the lane markings (Borkar et al. (2009b)).

**Lane Detection in Parts**  Borkar et al. (2009b) takes a slightly different approach. They divide the ROI into left and right halves and apply a low-resolution HT in each. Once the approximate line positions are found in each half of the image, the lines are split into discrete points and horizontal search windows are deployed at each point to find a more accurate position of the lane markings. This is followed by applying a HT to fit a line to all discrete points. One problem with such method where each half of the image is considered separately is, when we are trying to find a road lane (a collection of two lines), if the detection of each line is done separately, there may be no correspondence between the lines detected in the two images when combined together. Moreover, in this case, HT is used twice which makes it less useful for GPU/FPGA computing, for the reasons mentioned earlier.

**Randomized Line Detection**  To avoid the inherent shortcomings in HT, Chen et al. (2001) has proposed an alternate approach for line detection namely, Randomized Line Detection (RLD) algorithm. This approach involves picking up three random edge points in the edge-detected image and finding if the candidate lines detected are truly the lane markings on the road. Advantage of their approach
as opposed to HT is that it doesn’t need an additional storage for accumulator array. However their algorithm is designed for a sequential implementation on a CPU and not quite suitable for parallel implementation on GPUs/FPGAs.

**Color Segmentation** There are also color based segmentation methods that use color information in the image and extract the lines. Chiu et al. (2005) and Ma et al. (2010a) present such methods. However, these methods work based on detecting all the pixels that are within a predefined color-space to represent the lane markings. But the street lighting conditions especially in the night may vary significantly which may in turn affect the color in which the lane markings appear. This leads to questions about the robustness of such methods.

**Steerable Filter** Detailed comparison of more lane detection methods is presented in McCall et al. (2006). It also presents a new method for detecting the position of lane markings in the image using Steerable Filters. These filters help in changing the direction in which the gradients are to be evaluated. In case of straight line markings on the highways, a 1D Gaussian kernel would be sufficient. But if there are curved markings (circular reflectors) as the road curves ahead of the car, it is better to use a filter that can change its direction so as to appropriately detect the markings. In their paper, formulae are given for evaluating the 2D Gaussian terms $G_{xx}, G_{yy}$ and $G_{xy}$. Using these terms, one can evaluate $\theta_{\text{min}}$ and $\theta_{\text{max}}$ that determine the directions in which the minimum and maximum responses of the filter to the lane markings occur. Hence, it helps much better in detecting the omni-directional line markings. However, in the current work, we mainly concentrate on the straight line markings and hence, it is out of scope of the current work.

**Our Method** In the current work, we propose a combination of particle based and gradient based approaches for detecting the lines. We use several random particles, each representing the characteristic of a line with the parameters ($x$-intercept, slope) and assign a weight to each particle based on the gradient image calculated. In this method, we do not need to maintain an accumulator array as in the case of HT, which reduces the memory transfers between GPU/FPGA and the Host CPU. Secondly, using independent particles is quite convenient for parallel processing because thousands of particles can now benefit from the potential of GPU/FPGA to process them parallely. We use a gray-scaled image before lane
detection step so that our method is robust enough to work in any street lighting conditions without having to depend on the color segmentation.

Lane detection is a time consuming step. Hence a combination of lane detection step once every few frames and lane tracking for rest all frames is a more efficient method. Presented in section 3.2 is a literature survey of lane tracking methods.

3.2 Lane Tracking Methods

Lane Tracking is primarily used to enhance the computation efficiency of the lane detection algorithm by maintaining the previous information of how the states have evolved over time so as to have an estimate of the future states. Usually this involves a prediction step and a measurement step. In case of Lane Tracking, prediction step involves moving the detected lines by a certain amount in the image, based on ego-vehicle velocity or by making some assumptions. In measurement step, a new measurement is obtained which will then be used to correct the predicted lane marking positions. Significant research has already been done in Lane Tracking. The most frequently used lane tracking filters are the Kalman Filter and Particle Filter. Sections below give a brief overview of literature in this direction.

3.2.1 Kalman Filter

In Liu et al. (2011), after lane detection is done, a Kalman Filter is used to track the lanes. An ego vehicle lateral offset is predicted and the lane lines are selected based on this predicted value. Once the lines are selected, the ego vehicle lateral offset is updated. Though the performance of their algorithm was reported to be providing up to 94.34 Hz on a PC (based on AMD Athlon64 3000+), there are several strong assumptions made about a constant ego vehicle lateral offset, lane width and the width of lane markings. This raises questions about the robustness of the algorithm on different situations. A strong model always has its own limitations as compared to weak model which makes very less assumptions and derives the required values as the computation progresses.

In McCall et al. (2006), lane detection is done using steerable filters (mentioned in the previous section) followed by lane tracking using Kalman Filter. It
constructs lane tracking algorithm in two steps as with most other works - prediction and measurement update. In prediction step, it uses lateral position of the vehicle relative to lane markings, lane angle, curvature of the road, steering angle and the lane width to update the states (position and orientation of lane markings) of the Kalman filter. Before prediction step, all the required parameters are derived using certain methods mentioned in the paper. The prediction step is followed by a measurement update where a measurement vector containing vehicle position, lane angle and lane width is updated using Hough Transform (HT) and lane detection statistics. This method makes no assumptions regarding the width of the road or vehicle velocity but derives everything, which makes it robust on a CPU.

However, the use of HT in Lane Tracking step is something different from other works. It can help greatly in increased accuracy but at the cost of penalty on computation efficiency if their algorithm were to be implemented on a GPU because, HT needs an accumulator array with large memory that needs to be exchanged between host CPU and the GPU. Hence, the very motive of improving the Lane Detection algorithm’s efficiency using Lane Tracking technique will be lost. The throughput that was reported for their system is around 30 Hz. Borkar et al. (2009a) takes a slightly different approach that doesn’t use HT for Kalman filter update. In their work, after detecting the lane location, an outlier elimination step is applied using Random Sample Consensus (RANSAC) algorithm. This is followed by lane tracking by a Kalman filter that uses position, velocity, orientation and angular velocity of the lane markings relative to the initial image frame. In the measurement step, if the lines are not accurately detected, the lines from prediction step are still retained. Though their method might suffer from inaccuracies if the prediction results are kept intact sometimes, the difference in the lane’s position and orientation is not very large in practice between consecutive frames. Hence, this is a faster method as compared to the previous one.

Kalman Filter provides an optimal solution as long as the probability distribution of the states to be estimated are linear and the noise is Gaussian but it fails when the problem is non-linear. In this regard, there is a modified version of the Kalman Filter known as Extended Kalman Filter (EKF) used in Meuter et al. (2009) and Zhao et al. (2012) especially for Lane Tracking problem. EKF basically retains the first non-linear term in Taylor expansion of the function (Arulampalam et al. (2002)). It could also retain more terms to better simulate a non-linear problem at the cost of increased complexity which is not desirable especially
when performance is the criteria. Hence, Monte-Carlo simulation based Bayesian Methods known as Particle Filters are quite efficient in capturing the non-linear dynamics of the problem. Given in the next section is a short review on such methods applied to Lane Tracking problem.

### 3.2.2 Particle Filter

In Liu et al. (2010), lane detection is done using an improved version of HT, called Statistical Hough Transform (SHT) which uses all the pixels in the image unlike the standard HT that uses only few pixels leading to ambiguity in the actual edges. It combines this lane detection technique with a Particle Filter for lane tracking. The control points of a cubic spline that is fit to lane markings are the states of the particle filter that are tracked. *Stratified Resampling* is used to resample the particles based on their weights. And finally, the high weight particles after resampling are chosen as the detected lines. This algorithm was implemented in MATLAB on Intel Core 2 Duo (2.2GHz) machine. However, the throughput of lane tracking algorithm is as low as 1.0 Hz.

In Ruyi et al. (2011), a bird’s view of the image is obtained and the edges are detected. After edge detection, a Real Orientation Distance Transform (RODT) is applied to the image to assign to each pixel, the value of the distance to the nearest edge. This helps in identifying the boundary points that make the lane markings on the road. The coordinates of these boundary points are used to initialize the particle filter for actual lane detection. However, to track lane markings, neither Kalman nor Particle filter was used but a novel tracking technique was suggested by considering the coordinates of the boundary points as the states that need to be updated for every new frame by taking ego-vehicle motion model into account to move these boundary points in each consecutive frames. The algorithm was implemented on Intel dual-core E5300 PC (2.5 Ghz) and the frame rate obtained for lane tracking is around 15.87 Hz.

One of the main aspects of using Particle Filtering techniques is to know which motion model accurately fits a given problem. Most of the works in the literature either assume a motion model or obtain the accurate information on vehicle’s motion using inertial sensors. However, there are also learning approaches in which neither assumptions are made nor measurements are taken to determine the motion model but are deduced from the previous states of the tracked parameters. One such procedure is suggested in Gopalan et al. (2012). It uses around
200 particles to track the lane markings with a separate particle filter dedicated to each of the lane markings. Based on the variations in the state parameters assigned to each of the particles, deductions are made as to what might have caused the variation in these parameters. Based on these deductions, the motion model is finally deduced for the vehicle. The lane tracking algorithm in their case was reported to be working on 240x320 images at 25Hz on a 4 GHz processor.

While the major advantage of a particle filter being its ability to capture non-linearity, Kalman Filter can provide optimal solution if a part of the solution is linear. Particle filter can also reach such a solution but at the cost of including more particles which in turn can increase the time of computation. Nieto et al. (2012) provides a very innovative approach to derive the advantages in both Particle Filter and Kalman Filter. In this, the lane tracking problem is split into two separate sub-problems. The states that are linear are processed by Kalman filter while Particle Filter estimates the non-linear states. This algorithm was tested on Intel Atom CPU N270 (1.6 GHz) and the performance obtained is up to 30Hz.

Our Method Though there is significant amount of literature available on Lane Detection and Tracking, it is almost rare to find an implementation especially aimed at heterogeneous, high-performance platforms like GPUs and FPGAs which have the potential to replace the ECUs in the future electric vehicles. In the current work we use Particle Filter to perform lane tracking after the lane detection step. While we follow a weak model (no assumptions about the lane width, lane marking size), which makes our algorithm quite robust in different road conditions, our main aim is to implement the particle filter on heterogeneous devices (GPUs and FPGAs). The prediction and measurement update steps of the particle filter are performed on such devices supported by OpenCL, while the OpenCL host is an Intel Core2 Quad Processor Q9550 (2.83 GHz). With this set-up, our particle filter processes the images at up to 104 Hz on NVIDIA GTX 660 Ti GPU, up to 78 Hz on NVIDIA GTX 285 GPU and up to 27 Hz on Altera Startix V FPGA. Complete detail on the method and implementation is provided in chapter 4.
Chapter 4

Method and Implementation

In this chapter, the overall implementation of the current work is presented in detail. This work can be divided into two parts:

1. Developing lane detection and tracking system for CPU (section 4.1).
2. Porting the CPU code to OpenCL and testing it on heterogeneous hardware (GPU and FPGA) (section 4.2).

4.1 Developing lane detection and tracking system for CPU

Figure 4.1 is a schematic that gives a brief overview of the functions involved in lane detection and lane tracking system. Following sections provide details of each of these blocks.

4.1.1 Preprocessing the Image

Graycaling and Selecting the Region of Interest

First step involves gray-scaling the color image captured by the camera. The gray-scaled image is then cropped to get the Region Of Interest (ROI) so as to avoid the regions where there are lesser chances of finding road lanes. The actual road-region visible to the camera is significantly smaller as compared to the other regions in the image, like the pavements or trees on either side of the road, the sky
and other vehicles ahead on the road. Hence, selecting ROI gives a better estimate of the detected lanes and significantly reduces the computation time. Figure 4.2 shows an example of ROI.

![Figure 4.1: Lane Detection and Tracking Schematic](image)

**Figure 4.1:** Lane Detection and Tracking Schematic

**Figure 4.2:** Example of ROI selection for lane detection.

**Edge Detection**

Gaussian smoothing is done to the selected ROI to obtain a blurred image to reduce noise, followed by a Sobel filter to detect the edges in the image. The Sobel
filter used here is from the OpenCV library. Figure 4.3 shows a snapshot of the detected edges.

![Figure 4.3: Edges detected using Sobel filter](image)

### Gradient Matrix Computation

After the edges are detected, a gradient matrix is computed by taking the difference in pixel values of the two adjacent pixels at a time along each row using the formula given Equation 4.1.

\[
G_x(row, col) = |p(row, col) - p(row, col - 1)|
\] (4.1)

\(G_x\) represents the gradient matrix and its subscript denotes that the gradient in being taken along the x-direction. Here, \(p(row, col)\) is the gray-scale value of each pixel of the matrix with range \(row \in (1, \text{num}_{\text{rows}})\) & \(col \in (2, \text{num}_{\text{cols}})\). Since we are interested in finding the road lines that are oriented closer to the vertical, taking a horizontal gradient would suffice in this case. However, one can also take vertical gradient \(G_y\) to evaluate \(\sqrt{G_x^2 + G_y^2}\) to obtain a better gradient estimate. We consider only the absolute values of the differences in Equation 4.1.

Once the gradient matrix is computed, a simple thresholding step is introduced to set any element of the gradient matrix that is less than a certain percentage of the element with the maximum value in the entire matrix to zero. Rest of the matrix elements are left intact.

### 4.1.2 Lane Detection

After preprocessing an image, a conditional branching checks if the image has to be sent to a "Lane Detection" or a "Lane Tracking" block as shown in Figure 4.1. Since lane detection is a computationally expensive step, we use lane tracking to keep track of the lines that are once detected. Hence, a lane detection is performed on every \(L^{th}\) image after skipping an \((L - 1)\) number of images which all go through
the lane tracking process described in subsection 4.1.3. In this section, we describe the **Lane Detection** block which is shown in the Figure 4.4.

![Lane Detection Diagram](image)

**Figure 4.4: Lane Detection Schematic**

**Line Sampling and Weight Evaluation**

To be able to detect the lines in the image, several thousands of line models are randomly sampled to occupy the ROI. We then assign weights to each line model and extract the lines with the highest weights to represent the lines on the actual image. Procedure is described below.

To define each line, we use a slight variation of the slope-intercept form. Figure 4.5 shows how each line is created on the image plane. Since image exists as a matrix of pixel values, we represent each line as a structure that maintains the coordinates \((X_l(num\_rows), Y_l(num\_rows), \theta_l)\) that define a line \(l\). Notice that \(X_l\) and \(Y_l\) are arrays, each of length equivalent to the number of rows in the image, while a single value \(\theta\) is associated to each line. The red line in the figure shows a single line while the blue dots represent the discretized points at which the coordinates are maintained in the structure that defines a line. Also notice that the positive directions of X and Y axes that are used in this work are shown in the figure. For lane detection, we populate the image with thousands of such lines randomly.

Random sampling here is done by creating lines with X-intercept falling anywhere on the horizontal edge of the image and the slope \(\theta\), anywhere between 0 and \(\pi\). Once the lines are created, the weight of each line is evaluated using Equation 4.2.

\[
w_l = \sum_{row=1}^{num\_rows} \sum_{col=X_l(row)+N}^{X_l(row)+N} G_x(row, col) \tag{4.2}
\]

Here, \(w_l\) is the weight of line \(l\), \(N\) is the neighborhood used to decide how many pixels in gradient matrix on either side of the line \(l\) will be considered for
weight evaluation. \( x_{i}(\text{row}) \) represents the x-coordinate from which the line passes at the y-coordinate represented by \( \text{row} \). Summation is carried out to add up all the gradient values within the neighborhood along each row and one more summation to add up over all the rows to obtain the complete weight of a line.

Once the weight is estimated for all the lines using this procedure, they are sorted according to their weights. The lines with highest weights are naturally chosen to be the lane markings.

**Distance Criteria**

When detecting a lane on the road, each of the two lines might be represented by multiple, closely spaced, highest weight line-models while completely neglecting the second line. This issue is depicted in Figure 4.6. Hence, a criteria is introduced to ensure that the line-models are at least separated by a predefined distance.

Usually, a distance criteria between 30-60% of the width of the ROI is used to select only those lines that are at least separated by this distance criteria. Figure 4.7 shows the output of the algorithm after introduction of the distance criteria.
Chapter 4. Method and Implementation

Figure 4.6: Output (green lines) of lane detection algorithm before introducing the distance criteria into the code. The lane marking on the left side is completely neglected and only the right one is represented by both the green lines.

Figure 4.7: Output after introducing the distance criteria. Now both the desired lines are detected by the algorithm accurately.

Output after introducing the distance criteria. Now both the desired lines are detected by the algorithm accurately. Assumption of this algorithm is that, once the ROI is obtained, the lane markings will be of the highest gradients. Sometimes, there might be a difficulty in detecting the lanes accurately if the ROI is not selected correctly or if there are other very high contrast members within ROI, like road sign boards or vertical painted frames/walls on either side of the road, that do not represent the road lanes. But this algorithm detects the lines fairly accurately across a range of images captured.

4.1.3 Lane Tracking

As mentioned in section 2.2, a lane detection step is a computationally expensive stage since it discards all the information regarding the lane markings detected in the last image and re-evaluates the lanes afresh. A better solution would be to retain the information of the states that describe a line from the previous stage to make corrections to the positions of the lane markings in the new image based on the new measurements obtained. We use a Particle Filter for lane tracking for which the implementation is presented below.

Figure 4.8 shows the major blocks contained in the particle filter algorithm.

In the next section, we describe the particle filter developed for tracking a robot moving in a 2D plane which is then extended to the final lane tracking algorithm in section 4.1.3.2. We also present how each step in these cases relates to the blocks shown in the Figure 4.8.
4.1.3.1 Particle Filter for Tracking a Robot in 2D

Consider a scenario shown in the Figure 4.9a where a robot is placed in a workspace of size 100x100 and the robot doesn’t know its position and heading direction. In order to estimate these parameters through particle filter, the entire workspace is initially populated with samples of 2D points, while each point represents a particle with the similar parameters of the robot that are to be estimated; in this case, position and heading direction. The workspace after being populated with particles is shown in the Figure 4.9b. Each particle and the robot are considered to be maintaining a set of independent 2D coordinates \((x_i, y_i)\) and a variable \(\theta_i\), the heading direction in which the particle (or the robot) is assumed to be moving. While populating the particles in the workspace, each particle is assigned a coordinate randomly (Monte-Carlo approach) within the limits of the workspace boundaries and a heading direction anywhere in the interval \((0, 2\pi)\) rad.

After initializing the particles in the workspace, Algorithm 4.1 is performed. The actual purpose of using particles is to represent the robot’s unknown position and heading direction with the known parameters of the particles that have high weights. The higher the weight of a particle, the better it represents the robot. For each iteration, the Robot is moved (using \textit{MOTION\_UPDATE} routine) by a distance \(\Delta d\) along the new heading direction \((\theta + \Delta \theta)\) and a measurement is obtained (from \textit{MEASUREMENT\_UPDATE} routine) by evaluating the distances of the robot from each of the four corners of the workspace. This is repeated with all the particles as well, except that the values of \((\Delta d, \Delta \theta)\) have to be slightly different for each particle. To maintain this difference in motion for different particles, a noise variable \textit{motion\_noise} is introduced because, at any given iteration, we would like to retain a collection of high weight particles instead of just the best particle, around the robot so that in the next iteration, if the robot moves to a
new position, other particles from the collection can represent the robot. Here, the measurement noise is introduced to account for any measurement uncertainties in real-world applications.

**Algorithm 4.1** Particle Filter for Tracking a Robot

\begin{algorithm}
1: for \( i = 1 : \text{iterations} \) do  
2: \textit{MOTION\_UPDATE}(\textit{Robot}, \Delta d, \Delta \theta, 0.0)  
3: \textit{MEASUREMENT\_UPDATE}(\textit{Robot}, \textit{measurement\_noise})  
4: for \( p = 1 : \text{num\_particles} \) do  
5: \textit{MOTION\_UPDATE}(\textit{particle}(p), \Delta d, \Delta \theta, \textit{motion\_noise})  
6: \textit{MEASUREMENT\_UPDATE}(\textit{particle}(p), \textit{measurement\_noise})  
7: end for  
8: \texttt{*particle\_new} \leftarrow \textit{RESAMPLE}(\texttt{*particle})  
9: \texttt{*particle} \leftarrow \texttt{*particle\_new}  
10: end for
\end{algorithm}

**Motion Update** The \textit{MOTION\_UPDATE} subroutine in Algorithm 4.1 is constructed based on Equation 4.3.

\[
Pr(X_t|Y_{t-1}) = \sum_i Pr(X^i_t|X^i_{t-1}, Y^i_{t-1}) \times Pr(X^i_{t-1}|Y^i_{t-1}) \times \Delta X_{t-1} \tag{4.3}
\]
which gives a prediction of how the probability distribution of the new state $X_t$ might look like, using the values of the old state $X_{t-1}$. Hence this subroutine is also known as prediction step. In this case, each of the particle’s state-space contains three parameters, $X^i_t = (x^i_t, y^i_t, \theta^i_t)$, where index $i$ identifies each particle.

**Measurement Update**  A motion update may not give an accurate probability distribution because it is based only on the old measurements. Hence, every motion update is followed by the measurement update a.k.a subroutine MEASUREMENT_UPDATE in Algorithm 4.1. Mathematically, this step is represented by the Bayes rule given in Equation 2.6. To be able to construct this equation, we need to evaluate the terms likelihood, prior and evidence as described before. Prior is the probability distribution obtained after the prediction update. The numerator of Equation 2.6 which is likelihood $\times$ prior is obtained from Equation 4.4, which gives the weight assigned to each particle by comparing its state after the motion update to the actual robot’s states obtained from the new measurement update. Such weights are assigned to each particle by fitting its state variable values $X^i_t$ to a Gaussian function given in Equation 4.4 with mean value $\sigma$ as the robot’s position and the covariance $\mu$ being a noise value called measurement_noise in the Algorithm 4.1.

$$w_i = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2}\left(\frac{X^i_t - \mu}{\sigma}\right)^2}$$

(4.4)

To obtain the evidence in Equation 2.6, we finally sum up the weights over the entire range of particles, $N_p = NUM\_PARTICLES$. Hence,

$$evidence = \sum_{i=1}^{N_p} w_i$$

(4.5)

The particles which are closer to the robot end up having a higher probability value. After every measurement update, the probability distribution of the particle-states becomes narrower as compared to the prediction update because the estimate of the states becomes more accurate after measurement update.

**Resampling**  After obtaining the new probability distribution for the current time step, a RESAMPLE subroutine is used to obtain a new sample of particles chosen from the old set based on their weights values. Algorithm 4.2 presents this subroutine.
Algorithm 4.2 RESAMPLE

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

Here, \( N_p = NUM\_\text{PARTICLES} \), \( idx \) is an index drawn randomly from the particle indexes. The weight \( w_{\text{max}} \) belongs to the particle with maximum weight after the measurement update. Hence, a variable known as \( \beta \) is assigned a random weight which is within \( 2 \times w_{\text{max}} \). This algorithm ensures that any particle with weight less than the value of \( \beta \) is skipped and all other particles are kept in the new set.

While the new set contains the same number of particles as the old set after the resampling step, there can be multiple copies, in the new set, of some particles from the old set, while some of the particles can be completely missed out in the new set even if they have high weight. Usually, the higher the weight of a particle, the higher the probability of multiple copies of that particle in the new set.

This cycle of motion update, measurement update and resampling continues as long as the robot’s tracking is to be performed. The results for this subsection are presented in section 5.2. More ideas on this particle filter are presented in Thrun, S. (2013).

4.1.3.2 Particle Filter for Tracking Road-Lane Markings

As described earlier (subsection 4.1.3), to enhance the computation efficiency, after detecting the lines (subsection 4.1.2), use of a particle filter to track the detected lines can be beneficial. When a particle filter that is described in subsection 4.1.3.1 is extended to tracking the line-markings on the road, the particles should be modified to contain information about line samples rather than simple points as in subsubsection 4.1.3.1. The state-space of each particle now contains
\[ X_i^t = (x_i^t(N_r), y_i^t(N_r), \theta_i^t), \]
where \( i \) identifies the line, the arrays \( x_i^t \) and \( y_i^t \) each of size \( N_r = \text{num\_rows} \) (number of rows of the ROI of the image) contain all the \( x \) and \( y \) coordinates of the points that together build a line. \( \theta_i^t \) is the orientation of line \( i \) w.r.t the \( x \)-axis of the ROI plane.

Since a lane detection step usually initializes the particle filter, the lines are created as described in section 4.1.2 and the lane markings are detected before the particle filter step. Along with the two lines (Figure 4.7) that best represent the lane markings, a number of other line samples are also maintained for particle filter to track the lines. Henceforth, the two lines will be called the best-lines and the rest of the lines, the good-lines. Figure 4.10b depicts the good-lines (blue) and the best-lines (red).

Algorithm 4.3 summarizes the method for applying the particle filter for tracking the lines and this algorithm has great similarity to the particle filter described for simple points in 2D plane in Algorithm 4.1.

Algorithm 4.3 Particle Filter for Lane Tracking

1: for \( i = 1 : \text{num\_good\_lines} \) do
2: \( \text{MOTION\_UPDATE}(\text{good\_lines}(i), \Delta \rho, \Delta \theta, \text{motion\_noise}) \)
3: \( \text{MEASUREMENT\_UPDATE}(\text{good\_lines}(i), \text{measurement\_noise}) \)
4: end for
5: \( \ast\text{good\_lines\_new} \leftarrow \text{RESAMPLE}(\ast\text{good\_lines}) \)
6: \( \ast\text{good\_lines} \leftarrow \ast\text{good\_lines\_new} \)

In this algorithm which repeats for each new frame of the video stream, MOTION\_UPDATE routine shifts each line by \( \Delta \rho \) and rotates by \( \Delta \theta \), while adding a motion\_noise to spread the distribution of the good\_lines by an uncertainty
about the best lines. The MEASUREMENT_UPDATE subroutine evaluates the new weights of the good lines following a similar procedure described in the previous section for simple points. The MEASUREMENT_UPDATE compares the $x$-intercept and orientation of each good line with the nearest best line and assigns a weight along the Gaussian probability distribution function given by Equation 4.4 based on how closely a good line compares to the best line. This is followed by RESAMPLE subroutine similar to Algorithm 4.2.

4.2 Parallelizing the Algorithm for OpenCL

4.2.1 OpenCL Software Implementation

This section provides the details of porting the lane detection and tracking code from CPU implementation (in C/C++) described in section 4.1 to OpenCL.

Figure 4.11 presents the schematic followed for OpenCL implementation of the code on each of the GPUs. Comparing this schematic to the one presented in Figure 4.1 for the pure CPU implementation, we see that the overall workflow remains almost identical except a few changes: a new block for OpenCL initialization (block in red color) and a few modifications in the blocks of "Lane Detection" and "Lane Tracking".

In OpenCL initialization block, OpenCL platform, context, command queues and program are created which will then be used by both lane detection and lane tracking algorithms for executing the kernels in parallel on the OpenCL devices. This block is implemented in the host side code using the OpenCL Platform APIs described in section 2.3.2.1. In each of lane detection and lane tracking blocks, the host code contains the OpenCL Runtime APIs that are used to enqueue the buffers and execute the device code (kernels) written in OpenCL Programming Language.

Parallelizing every block of code written for CPU is not possible because some blocks have to be implemented sequentially. The kernels are thus created for executing some blocks of code wherever the algorithm enables parallel processing on OpenCL devices. When creating the lines in the block "Sample the Lines" in Figure 4.4, thousands of lines are processed independently. In case of CPU implementation, this is achieved using a for loop repeating for each of the lines in a sequential manner. However, no line's initialization process depends on the
other lines created before. Hence, it is a potential parallel block which can benefit from the OpenCL devices. Hence the for loop is replaced by OpenCL’s NDRange that executes several lines parallelly as independent threads in an SIMD (Single Instruction Multiple Data) fashion.

Blocks titled ”Motion Update” and ”Measurement Update” in the Figure 4.8 also allow parallel processing each line independent of the others. Hence, these two blocks were combined into a kernel. Thousands of threads following the instructions written in a kernel can now execute parallelly. However, the block titled ”Resampling” needs comparing the weights of several lines. Hence, this is not a potential member for parallelization. The kernels that run on OpenCL devices are as follows:

- **Kernel-1** creates lines and initializes their coordinate data structures in Lane Detection block.

- **Kernel-2** performs Motion Update and Measurement Update in Lane Tracking block.

Figure 4.11: Schematic used for running Lane Detection and Tracking using OpenCL

Kernels 1 & 2 are the *Device Codes* mentioned in each of the lane detection and lane tracking blocks respectively in Figure 4.11. Based on the conditional
branching tested by the block "frame%L = 0?", the process either enters lane detection or lane tracking block based on the frame number frame and manually chosen number L. Based on which block it enters, the process executes either Kernel-1 or Kernel-2 while both of these make use of the same platform, devices, context, commandQueue and program that was created in the red block. In both the lane detection and tracking blocks, a certain set of protocols are followed for execution of the respective kernels. This is depicted in Figure 4.12 for GPUs and in Figure 4.13 for FPGA.

The only difference between the OpenCL execution schematic for GPUs and FPGA is that the kernels used in case of GPUs are compiled by OpenCL during the runtime whereas the kernels that are to be run on FPGA have to be compiled into hardware design image files known as Altera Offline Compiler Executable files (*.aocx) before the execution begins. This additional step is indicated in Figure 4.13 for FPGA. Compiling such *.aocx files from kernels using Altera Offline Compiler (AOC) takes a longer duration, sometimes even longer than 6 hours based on the complexity of the kernels used. Hence it is a time-consuming step in case FPGA. These *.aocx files are then loaded on to the FPGA board during the runtime to configure and allocate FPGA resources before the kernels execute.

![Figure 4.12: Schematic for OpenCL Execution on GPUs.](image-url)
Notice that the OpenCL initialization block (in red) in Figures 4.11, 4.12 and 4.13 are identical. When the execution process enters one of the lane detection or lane tracking blocks, the memory objects or buffers are created using the OpenCL Runtime API `clCreateBuffer`. This is followed by the API calls `clEnqueueWriteBuffer`, `clSetKernelArg`, `clEnqueueNDRangeKernel` and `clEnqueueReadBuffer`. The cycle marked in green colored arrows in the Figure 4.12 continues in ”Lane Tracking” block for \((L - 1)\) image frames and repeats this process once in ”Lane Detection” block, every \(L^{th}\) frame. However, the buffers that are created depend on which block this cycle is performed in.

### 4.2.2 Hardware Architecture of OpenCL Devices Used

#### 4.2.2.1 GPU Specifications

Table 4.1 presents some of the technical specifications, especially the details that affect an OpenCL implementation. We use two GPUs, both from NVIDIA. The **GTX-660 Ti** is a more powerful GPU than **GTX-285** and also consumes much less power per performance delivered (GFLOPS/w in the table). Here, the cores of the GPUs are organized into groups - each Streaming Multiprocessor (SM) is considered as a **Compute Unit** while several threads (**Work Items**) within an SM are organized into **Warps** of size 32 each. A warp is considered as a single **Work Group** by OpenCL. Several SMs are organized into one group known as **Thread Processing Cluster** (TPC). One can notice that **GTX-660 Ti** delivers almost thrice the raw
computing power (GFLOPS) as compared to the other GPU. These characteristics will be more evident in the results presented in chapter 5.

Table 4.1: Key Hardware Specifications for GPUs Used

<table>
<thead>
<tr>
<th>Property</th>
<th>NVIDIA GeForce GTX 285</th>
<th>NVIDIA GeForce GTX 660 Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>NVIDIA Tesla</td>
<td>NVIDIA Kepler GK104</td>
</tr>
<tr>
<td>Thread Processing Clusters (TPCs)</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>Streaming Multi-processors (SMs) per TPC</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Threads per SM</td>
<td>1024</td>
<td>2048</td>
</tr>
<tr>
<td>Streaming Processors (SPs) per SM</td>
<td>8</td>
<td>192</td>
</tr>
<tr>
<td>Total Cores</td>
<td>240</td>
<td>1344</td>
</tr>
<tr>
<td>Clock Speed (MHz) of each Core</td>
<td>1476</td>
<td>1124</td>
</tr>
<tr>
<td>Single-Precision FLOPS/SP/Clock</td>
<td>3 (MADs)</td>
<td>2 (MADs)</td>
</tr>
<tr>
<td>GFLOPS (All Cores Together)</td>
<td>1062.72</td>
<td>3021.31</td>
</tr>
<tr>
<td>Max. Power (w)</td>
<td>204</td>
<td>150</td>
</tr>
<tr>
<td>GFLOPS/w</td>
<td>5.21</td>
<td>20.14</td>
</tr>
<tr>
<td>Host Interface</td>
<td>PCIe 2.0 x 16 lane (8GB/s)</td>
<td>PCIe 3.0 x 16 lane (15GB/s)</td>
</tr>
<tr>
<td>MAX_WORK_ITEMS</td>
<td>512 × 512 × 64</td>
<td>1024 × 1024 × 64</td>
</tr>
<tr>
<td>MAX_WORK_GROUP_SIZE</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>MAX_GLOBAL_MEMORY</td>
<td>1,073,020,928 (1GB)</td>
<td>2,146,762,752 (2GB)</td>
</tr>
<tr>
<td>MAX_CONSTANT_MEMORY (kB)</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>MAX_LOCAL_MEMORY (kB)</td>
<td>16</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 4.2 presents the relevant specifications of the CPUs used as OpenCL Hosts to the respective GPUs mentioned in the Table 4.1.
Table 4.2: Key Hardware Specifications for CPUs Used

<table>
<thead>
<tr>
<th>Property</th>
<th>CPU used with GPU1: GTX 285</th>
<th>CPU used with GPU2: GTX 660 Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model</strong></td>
<td>Intel® Core™2 Quad Processor Q9550</td>
<td>Intel® Xeon® Processor E3-1225</td>
</tr>
<tr>
<td><strong>Number of Cores</strong></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><strong>Clock Speed</strong></td>
<td>2.83 GHz</td>
<td>3.10 GHz</td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td>3,082,388 kB</td>
<td>20,519,708 kB</td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
<td>Ubuntu 12.04 LTS</td>
<td>Ubuntu 12.04 LTS</td>
</tr>
<tr>
<td><strong>OpenCL Version</strong></td>
<td>OpenCL 1.1</td>
<td>OpenCL 1.1</td>
</tr>
</tbody>
</table>

4.2.2.2 FPGA Specifications

Table 4.3 presents some of the useful hardware specifications in case of the FPGA used in the current work. Note that the properties for which the specifications are provided in this case are quite different from those presented for the GPUs in Table 4.1 because FPGA is a reconfigurable hardware, which means that the very hardware specification varies based on the hardware design created by compiling kernels. The hardware image file (the *.aocx file mentioned before in this section) that is loaded on to the FPGA in the runtime determines what percentage of the overall Logic Elements, Registers, Memory Blocks etc. will be used to run a particular kernel.

OpenCL parameters such as work-group size and number of Compute Units to be configured on FPGA can be specified through various attributes provided by Altera’s implementation of OpenCL. These attributes should be mentioned along with kernel function definition. Thus, the parameters like Maximum Cores, Maximum Threads in case of FPGA are bound to change unlike the fixed numbers provided for GPUs in Table 4.1. More details on how these attributes are specified and the corresponding resource usage on the FPGA board are provided in the results section. Note that the host CPU used for FPGA is the same CPU as the one used in case of NVIDIA GeForce GTX 660 Ti mentioned in Table 4.2.
Table 4.3: Key Hardware Specifications for the FPGA Used

<table>
<thead>
<tr>
<th>Property</th>
<th>Altera Stratix-V (5SGXA7) FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Elements</td>
<td>622,000</td>
</tr>
<tr>
<td>Registers</td>
<td>939,000</td>
</tr>
<tr>
<td>M20K Memory Blocks</td>
<td>2,560 (50 MBits)</td>
</tr>
<tr>
<td>Host Interface</td>
<td>PCIe 3.0 x 8 lane (10GB/s)</td>
</tr>
</tbody>
</table>

**Version of OpenCL Used**  Note that as of date, four major versions of OpenCL have been made available online: OpenCL version 1.0, 1.1, 1.2 and 2.0; in the current work, we have followed the specifications of OpenCL version 1.1 as the Altera’s FPGA (Stratix-V) used in the current work only comes with the support for this version of OpenCL.
Chapter 5

Results

This chapter provides results obtained in the current work. In section 5.1, results from our lane detection are provided using various test cases. Section 5.2 shows how a basic particle filter works for tracking a robot’s position in 2D workspace. This particle filter is later extended and applied to tracking lane markings on roads for which the results are explained in section 5.3 using five different datasets of varying road conditions and levels of difficulty. Finally, in section 5.4 the performance comparison for various OpenCL devices used in this work is provided.

5.1 Lane Detection

Using the lane detection system described in subsection 4.1.2, a code was developed primarily using C++ & OpenCV. Presented below are the results for various cases used for testing. In each of the cases presented below, 30,000 lines are used to initialize the lane detection algorithm. Table 5.1 contains the conditions used in each of the test cases and the time taken by the lane detection algorithm. In all the cases presented in this section, the region of interest (ROI) is chosen manually and the image is cropped accordingly before lane detection algorithm is run on an image. However, in the test cases provided in section 5.3, ROI is selected by the algorithm itself for the entire video sequence.

Case 1 shows a highway with one lane on each side of the road. Visibility of lane markings is high in this case and hence it is an easier test case. In Case 2, lane markings are tilted at a certain angle from the angle of vehicle’s approach. This case was chosen to demonstrate the robustness of lane detection algorithm
for varying lane angles. **Case 3** has older lane markings where the visibility reduces, making it slightly ambiguous to extract the markings as compared to Cases 1, 2. **Case 4** demonstrates the functioning of our lane detection algorithm for more number of lines to be detected in the image and thus suggesting that it can be used for multiple lane detection. However, note that our lane tracking algorithm (section 5.3) limits the tracking to a single lane of the road though our lane detection algorithm can detect multiple lanes. Finally, **Case 5** uses a chessboard to demonstrate the effectiveness of our algorithm in detecting multiple lines oriented more vertically while neglecting the horizontal lines which are very rarely found in the real road scenario.

**Case 1**

![Original image](image1.png)  ![After lane detection](image2.png)

*Figure 5.1: Test conditions given in Table 5.1.*

This test case was chosen to demonstrate the effectiveness of the lane detection algorithm in detecting the lines not just in white but also in other shades like yellow. This can be attributed to the grayscaling of the image before performing the lane detection step. Due to this, the algorithm doesn’t necessarily distinguish between the white and yellow lines but only considers the gradients at the edges of each line. Figures 5.1a & 5.1b show the test image before and after the lane detection. As given in the Table 5.1, a distance criteria of 6.0% is used to reject the lines that have higher weights but whose intercepts along the horizontal are closer than 6.0% of the total width of the image.
Table 5.1: Conditions used for various test cases of lane detection system.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Image Dimensions</th>
<th>Distance Criteria (%)</th>
<th>No. of Lines</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>79 x 298</td>
<td>6.0</td>
<td>4</td>
<td>5.45</td>
</tr>
<tr>
<td>Case 2</td>
<td>101 x 277</td>
<td>10.0</td>
<td>3</td>
<td>4.48</td>
</tr>
<tr>
<td>Case 3</td>
<td>96 x 575</td>
<td>30.0</td>
<td>3</td>
<td>5.44</td>
</tr>
<tr>
<td>Case 4</td>
<td>512 x 511</td>
<td>4.0</td>
<td>16</td>
<td>7.02</td>
</tr>
<tr>
<td>Case 5</td>
<td>810 x 809</td>
<td>4.0</td>
<td>9</td>
<td>7.81</td>
</tr>
</tbody>
</table>

Case 2

\( (a) \) Original image.  \( (b) \) After lane detection.

Figure 5.2: Test conditions given in Table 5.1.

This case is chosen to demonstrate the lane detection system for an image taken from a car that is at an angle larger than usual.

Case 3

\( (a) \) Original image.  \( (b) \) After lane detection.

Figure 5.3: Test conditions given in Table 5.1.

This test case shows that the lane detection system works even for a moderately worn out road, and for discontinuous markings in the middle of the road.
Case 4

Figure 5.4 shows the lane detection system applied to a generic set of interleaved black-white stripes. This is to demonstrate the robustness in detecting multiple lines in the image. One can notice that the second line (in red) from the right in the Figure 5.4b slightly deviates from the actual line in the original image. Since the lane detection system here uses a set of lines at random positions and angular orientations for initialization, it is often possible that there is no other line available in the neighborhood that approximates the actual line better than this line as per the chosen distance criteria (Table 5.1).

Figure 5.4: Test conditions given in Table 5.1.

Case 5

The test case in Figure 5.5 is chosen to show how the horizontal lines are neglected by the algorithm while only the vertical lines are detected. Since in the scenario of detecting the lines on the road, horizontal lines are found less often except at a few zones like the stop lines at the signaled junctions as shown in Figure A.1. Hence, the horizontal line detection is out of scope of the current work. Whether horizontal or the vertical lines that are detected depends on how the lines used for initialization are. While initializing, the image is considered to be in the $x$-$y$ plane where the origin is at upper left corner of the image while positive $x$-axis is along the horizontal and the positive $y$-axis is along the vertical. All the lines
used for initialization have their $x$-intercepts along the upper edge of the image while random orientations are chosen in the range $(0, 2\pi)$, measured with respect to the upper edge of the image. Hence enabling any line markings in the image that are not quite close to the horizontal.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5_5}
\caption{Test conditions given in Table 5.1.}
\end{figure}

\section{5.2 Particle Filter for Tracking a Robot in 2D}

In case of a Particle Filter used for tracking a robot in 2D, the workspace of the robot is initialized with numerous particles as previously depicted in the Figure 4.9b. After this initialization, Algorithm 4.1 is used according to which, in each iteration the robot and the particles are moved by a specified $(\Delta d, \Delta \theta)$ and a new measurement is obtained in the form of \textit{weights} assigned to each particle by \textit{MEASUREMENT\_UPDATE} subroutine in Algorithm 4.1. Figure 5.8 presents the results from the Particle Filter Algorithm 4.1 for six consecutive iterations. In each of these sub-figures, the green colored diamond shaped feature indicates the robot’s location along the direction defined by the corresponding axis while the rest of the points are the particles deployed by the Particle Filter to track the robot. In iteration 1, since no measurement is obtained yet, all particles have an equal (zero) weight. After the first measurement update (iteration 2), the weights are assigned to all the particles based on their proximity as well as their heading direction compared to those of the robot.
As can be seen in each of the iterations from 2-6, the red colored particles are assigned the highest weights and also they are closer to the robot’s location. As the particle distance from the robot increases, their weights reduce gradually to zero and hence, the red color turns to blue. This phenomenon is depicted in the Figure 5.6 for iteration 4. Additionally, Figure 5.7 shows the plot for the weight distribution of the particles along the $x$-direction for the same iteration. The green squared dot is the robot’s location along $x$ while the colors of the particles under the distribution curve represent the respective weights. It can be noticed that all the high weight particles (red) are located closer to the robot while the weights of the particles reduce as one moves farther from the robot. This is an indication that the Particle Filter is effective.

![Figure 5.6: Particles after resampling phase, for iteration 4.](image)

As per the Algorithm 4.1, in every iteration, the entire set of particles (the array particle) is passed through the RESAMPLE routine where the set of old particles are updated to a new set (the array particle_new), keeping more high-weight particles with sometimes, multiple copies (as described previously in sub-subsection 4.1.3.1) while choosing very less number of low weight particles. This re-sampling step is typically called the prediction update, where the distribution of the particle-weights is predicted before obtaining the next measurement update.
In Figure 5.8 the particles in iteration 2, after passing through the re-sampling phase, get updated to the new set of particles in iteration 3. Re-sampling of particles occurs for every iteration, hence always maintaining the particles that are closer to the robot’s new position and heading direction after moving to the new location, thus tracking the robot effectively.

5.3 Particle Filter for Tracking Road-Lane Markings

As described earlier, a Lane Detection step is a computationally expensive one. Hence, we use Lane Tracking to track the lines detected in section 5.1 by extending the particle filter for tracking a robot in section 5.2 to lane tracking.

We present the results for particle filter in this section for five different datasets that prove the robustness of our algorithm in various conditions on the road. The datasets used in this work are

1. TUM Night Dataset
Figure 5.8: In iteration-1, the robot (green colored diamond in each image) is surrounded by particles initialized with a zero weight. As the iterations advance, the distribution becomes narrower and more peaky at the robot, hence locating its position & thus, tracking it. Colors as calibrated by the color-bar indicate the particle weight value assigned to each particle by MEASUREMENT_UPDATE routine from Algorithm 4.1.
2. TUM Day Dataset

3. Caltech Cordova1 Dataset

4. Caltech Cordova2 Dataset

5. Caltech Washington Dataset

While the first two datasets mentioned above were captured in the surroundings of Technical University of Munich (TUM), the last three sets were obtained from the online resource (Mohamed, A. (2008)). While TUM datasets mostly contain the suburban roads in day and night time conditions with very less traffic, the Caltech Cordova1 and Cordova2 datasets demonstrates the working of our algorithm on urban roads with moderate traffic. Finally, Caltech’s Washington dataset proves the robustness of our algorithm even in very dark shadows on the road during daytime, when detection of the lanes becomes extremely challenging.

5.3.1 TUM Night Dataset

In Figure 5.9b, the green line boundaries represent the Region Of Interest (ROI) chosen for the TUM Night Dataset. As described in subsection 2.1.1, selecting a ROI involves several factors and in this work the ROI in every case is chosen manually. For this particular dataset of a suburban road, the lane design is quite simple with one lane on each side of the road. The video captured from the camera has a resolution of $640 \times 360$ and a frame rate of 25 frames per second (fps). The ROI is offset horizontally by 10% of the image width to avoid details of the image on the opposite lane and vertically by 70% of image height. Also, 20% of the image width is cropped on the right to avoid trees and road sign boards. The size of the ROI thus obtained for this dataset is $448 \times 108$. Computation time also depends on the selected ROI and hence varies for various datasets.

In Figure 5.9b, blue lines are the particles used by the particle filter, while red lines are the lines with high weights that represent the road lane markings. To improve the accuracy in locating the lane markings by particle filter, we perform a post-processing step as shown in Figure 5.9c. Once the blue marks are found as described Figure 5.9c, the final results of our lane detection and tracking algorithm look as shown in the Figure 5.10. Table 5.2 shows the general test conditions, many of which are used for most other test cases presented in the following sections. Any differences in these conditions are mentioned in the respective sections.
Table 5.2: General Test Conditions

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset Used</td>
<td>TUM Night Dataset</td>
<td>-</td>
</tr>
<tr>
<td>ROI</td>
<td>108 × 448</td>
<td>Dimensions (rows x columns) of the region of interest.</td>
</tr>
<tr>
<td>L</td>
<td>1000</td>
<td>Lane detection is performed every $L^{th}$ frame. Lane tracking is performed on all other frames. Refer to the symbol $L$ mentioned previously in the Figures 4.1 &amp; 4.11.</td>
</tr>
<tr>
<td>NUM_LINES</td>
<td>30,000</td>
<td>For lane detection.</td>
</tr>
<tr>
<td>NUM_GOOD_LINES</td>
<td>300</td>
<td>No. of particles used by Particle Filter (PF) for lane tracking.</td>
</tr>
<tr>
<td>NUM_BEST_LINES</td>
<td>2</td>
<td>No. of Line Markings to be detected</td>
</tr>
<tr>
<td>BEST_LINE_DIST</td>
<td>60</td>
<td>Distance Criteria (subsection 4.1.2): Minimum distance (in % of image width) to be maintained between detected lines.</td>
</tr>
<tr>
<td>TURNMAX</td>
<td>1.50</td>
<td>Max. turning angle for moving any line used by PF. Value in Radians.</td>
</tr>
<tr>
<td>DISTMAX</td>
<td>30.00</td>
<td>Max. distance for moving any line used by PF. Value in no. of pixels along horizontal direction.</td>
</tr>
<tr>
<td>NEIGHBORHOOD</td>
<td>3</td>
<td>Number of pixels used on either side of a line to assign weight to the line.</td>
</tr>
<tr>
<td>DistNoise, TurnNoise</td>
<td>0.05, 0.05</td>
<td>Noise used to move lines by slightly different distance and orientation. These noises together make motion_noise in Algorithm 4.3</td>
</tr>
<tr>
<td>measurement_noise</td>
<td>0.05</td>
<td>Noise used in Algorithm 4.3 to account for real world noise in measurements.</td>
</tr>
</tbody>
</table>

One of the challenges in detecting the lane markings in the night is that, though the contrast of markings against dark road is good, there can be bright sign boards alongside the road as can be seen in Figure 5.10a or the bright light beams from the vehicles on the opposite side of the road as seen in Figure 5.10g that distract the gradient calculations. However, our algorithm still works decently in all such scenarios.

Figures 5.11 and 5.12 show how each of X-Intercept and the Slope - $\theta$ (as depicted previously in the Figure 4.5) for the detected lines (best-lines) vary over the frames of the video. This gives us an idea of the trajectory of the vehicle with respect to the lane markings on the road over the entire video sequence. In case of
(a) Original Image.

(b) Blue lines indicate several particles of the particle filter while red lines are the best particles chosen from the particles, to represent actual lane markings on the road. Here we use 300 particles to track the lane markings on the road.

(c) After the lane tracking step shown in Figure 5.9b, post-processing is done to introduce a horizontal search windows (orange lines) at a few discrete points along the best lines (red lines). Once a certain gradient is encountered along these search windows, the pixels are colored blue. This step increases the accuracy in locating the actual lane markings.

Figure 5.9
Figure 5.10: A few snapshots from the results for TUM Night Dataset. Figures in the left column are the original images while in the right are the lane-detected images. The blue colored markings are the lanes as detected by our algorithm.

X-Intercept, though the right (best_line1) and the left lane (best_line2) markings have different values, the difference of those two values remains almost constant for the entire video sequence. This means that the size of the lane remains almost constant over this video sequence. TUM Night Dataset was used to generate these plots.

Finally, Figure 5.13 shows how the likelihood values assigned to the best-lines vary across the frames of the video sequence. As indicated by the vertical axis, these values are too small because these likelihoods are normalized over all the
Figure 5.11: Vertical axis represents the X-Intercept made by the detected lines while horizontal axis shows the frame count of the video stream of 895 frames. Since there are two lines that are detected as shown in Figure 5.9b for each frame, there are correspondingly two different values of the X-Intercept for any given frame.

Figure 5.12: Vertical axis represents the Slope or $\theta$ of the detected lines while horizontal axis shows the frame count of the video stream of 895 frames. Since there are two lines that are detected as shown in Figure 5.9b for each frame, there are correspondingly two different values of the $\theta$ for any given frame.

particles used by the particle filter. However, the values shown here are only for the two best-lines in each frame.
Figure 5.13: This plot shows how the likelihood values assigned by the particle filter to the two best-lines in each frame vary across the video sequence of around 895 frames.

5.3.2 TUM Day Dataset

In Figure 5.14, a few snapshots are presented for the results from TUM day Dataset. The dimensions of the ROI selected in this case are similar to those chosen for TUM Night Dataset. Contrast of the lane markings is not very high in the day light scenario hence making lane detection in day to be different from night. However, our algorithm gives accurate results even in day time. Figures 5.14e and 5.14g show some curved lane markings, that are detected by our algorithm.

5.3.3 Caltech Cordova1 Dataset

This dataset was chosen to demonstrate urban road scenario with moderate traffic. For this case, the video captured from the camera has a resolution of $640 \times 480$ and a frame rate of 25 fps. The ROI selected in this case is significantly different from TUM Night Dataset. In Caltech Cordova1 Dataset, the lane marking layout is not a simple two lane suburban road scenario as in the case of TUM Night Dataset but a four lane urban road. Consider Figure 5.15a where the yellow lines separate the left and right side of the road. Each side of the road is further divided into two lanes by dotted white line markings. Also, camera placement is such that the lane in which the car moves is more or less centered in the image. Hence the cropping is done symmetrically on either side of the lane. For ROI, 30% of the
(a) (b) (c) (d) (e) (f) (g) (h)

Figure 5.14: A few snapshots from the results for TUM Day Dataset. Figures in the left column are the original images while in the right are the lane-detected images. The blue colored markings are the lanes as detected by our algorithm.

image width is cropped on either side of the image. Considering the sky in the upper half of the image and the vehicle hood in the lower half, 40% of the upper half and 30% of the lower half have been cropped in the image to obtain the final ROI of $256 \times 144$.

In case of Figures 5.15e and 5.15g, though there are no markings that define the lane, the lane tracking algorithm suggests the lane by continuing to show the old lane marking result until the lanes appear again on the road. In case of Figure 5.15i there is a vehicle ahead suggesting that the algorithm works in
moderate traffic as well.

![Figure 5.15:](image)

**Figure 5.15:** A few snapshots from the results for *Caltech Cordova1 Dataset*. Each original snapshot is followed by its corresponding lane-detected image. The green lined borders in the lane-detected images indicate the ROI while the red colored lines are the lane markings as detected by our algorithm.

### 5.3.4 Caltech *Cordova2* Dataset

This subset shows varying lane sizes and missing lines. The ROI chosen in this case is $256 \times 144$ and obtained using a similar procedure for cropping as described in the case of *Caltech Cordova2 Dataset*. In Figure 5.16a the lane is narrower than Figure 5.16c. However, using the lane tracking, the lane algorithm automatically takes care of these changes by moving the particles of the particle filter to the new position. Also notice that in the Figures 5.16c and 5.16i, the lane marking on the right is completely missing. However, the algorithm makes use of the gradients of the pedestrian pavements against the road to evaluate the ideal position of the lines.

### 5.3.5 Caltech *Washington* Dataset

This dataset shows the robustness of our algorithm in the presence of dark shadows on the road by the trees as well as moderate traffic on the urban roads. The ROI chosen in this case is $256 \times 144$ and obtained using a similar procedure for cropping
Figure 5.16: A few snapshots from the results for Caltech Cordova2 Dataset. Each original snapshot is followed by its corresponding lane-detected image. The green lined borders in the lane-detected images indicate the ROI while the red colored lines are the lane markings as detected by our algorithm.

as described in the case of Caltech Cordova1 Dataset. Shadows as can be seen in the Figures 5.17a, 5.17d and 5.17e make the gradient calculation for the detection of lane marking edges, quite challenging. But since our lane detection and tracking algorithm uses the summation of gradients along the entire length of a line to evaluate the line-weights, lane markings are still detected by using the high gradients in the non-shadowed sections of the line.

5.4 Performance of the OpenCL Implementation on Heterogeneous Hardware

In this section, a performance comparison is done for three OpenCL devices - NVIDIA GTX 660 Ti GPU, NVIDIA GTX 285 GPU and Altera Stratix V FPGA.

The kernels used for performing the computation on various devices are all written into a single kernel file that looks as shown in the Listing 5.1. While the kernel named createLineKernel is the Device Code for Lane Detection block in Figure 4.11, the kernel named goodLineWeightKernel is the Device Code in Lane Tracking block.
Figure 5.17: A few snapshots from the results for Caltech Washington Dataset. Each original snapshot is followed by its corresponding lane-detected image. The green lined borders in the lane-detected images indicate the ROI while the red colored lines are the lane markings as detected by our algorithm.

Notice that we use the OpenCL address space qualifiers \texttt{\_constant}, \texttt{\_global} for every argument to a kernel. These qualifiers determine where the data is located according to the OpenCL memory model described in chapter 2. The qualifier \texttt{restrict} must be used with every pointer in case of Altera’s OpenCL implementation. This is to ensure that the pointers are safely allocated in a unique memory location and other pointers are not assigned the same location.

Some of the functions required by the kernels are provided by \texttt{kernel_header.h} file. Notice that the arrays used to define the points on each line, \texttt{x[108]} and \texttt{y[108]} should be pre-allocated because OpenCL v1.1 does not support dynamic memory allocation within the kernels. The number 108 corresponds to the number of rows in the ROI considered for TUM Night Dataset. Hence, this number has to be manually varied each time a ROI dimension changes.

\begin{verbatim}
Listing 5.1: OpenCL Kernel File

    #include "kernel_header.h"
\end{verbatim}
typedef struct {
  float x[108];
  float y[108];
  float dist;
  float theta;
} lineStruct;

// --------------
// KERNEL 1
// --------------

__kernel void createLineKernel(
  __constant const struct createLineParams * restrict cline_params,
  __global const int * restrict img_grad,
  __global struct lineStruct * restrict line_output)
{
  size_t idx_dim1 = get_global_id(0);
  ...
  ...
}

// --------------
// KERNEL 2
// --------------

__kernel void goodLineWeightKernel(
  __global const lineStruct * restrict good_line,
  __global const lineStruct * restrict best_line,
  __global const int * restrict img_grad,
  __constant noiseStruct * restrict allnoises,
  __constant wtKernParams * restrict params,
  __global lineStruct * restrict good_line_out,
  __global float * restrict w_pf_out)
{
  size_t idx_dim1 = get_global_id(0);
  ...
  ...
}
Since lane detection is only performed once in \( L = 1000 \) frames as mentioned in Table 5.2, we are more interested in lane tracking code that is used more often (999 frames out of every 1000 frames). Hence, the performance of the algorithm depends largely on `goodLineWeightKernel` that performs prediction update and measurement update for each particle (line) used by the particle filter for every frame.

Consider the arguments of the kernel named `goodLineWeightKernel` shown in Listing 5.1. Taking the example of the test conditions (Table 5.2) used for TUM Night Dataset, the memories assigned to each of the kernel arguments are shown in the Table 5.3. Notice that the `good_line` is the pointer to the structures used to define 300 lines and this is the largest pointer in terms of memory size. Similarly, `img_grad` is the image sent to the kernel for each frame to perform the computation. These large memory pointers affect the memory transfers between host and the device significantly. While this code can further be improved by shifting the large memory objects from the slower global memory region to the faster, local memory region, it may not be possible in every case where the local memory blocks in the GPUs or FPGAs are not sufficient to hold such large pointers.

**Table 5.3: Memory Size used by Kernel Arguments**

<table>
<thead>
<tr>
<th>goodLineWeightKernel Kernel Argument</th>
<th>Memory Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*good_line</td>
<td>261600</td>
</tr>
<tr>
<td>*best_line</td>
<td>1744</td>
</tr>
<tr>
<td>*img_grad</td>
<td>193536</td>
</tr>
<tr>
<td>*allnoises</td>
<td>12</td>
</tr>
<tr>
<td>*params</td>
<td>32</td>
</tr>
<tr>
<td>*good_line_out</td>
<td>261600</td>
</tr>
<tr>
<td>*w_pf_out</td>
<td>1200</td>
</tr>
</tbody>
</table>

Using the setup described until now, tests were performed on all the three devices for varying number of particles. The performance graphs are shown in the Figure 5.18. While NVIDIA GTX 660 Ti GPU delivered the best performance, NVIDIA GTX 285 GPU stood next and Altera Stratix V FPGA is much slower as compared to GPUs. The general test conditions shown in Table 5.2 still hold good for the performance graphs shown in the Figure 5.18, except for a few changes mentioned in Table 5.4.
Chapter 5. Results

Figure 5.18: This graph shows the average frame rate Vs. Number of Particles for the three different OpenCL devices used in the current work. TUM Night Dataset is used for this comparison.

Figure 5.19

The frame rates shown in the Figure 5.18 are averaged over 899 frames. But to be able to visualize frame-by-frame performance, a comparison plot is shown for both the GPUs in Figure 5.19. The frame rates vary significantly for each frame. However, these rates oscillate about a value of about 75 Hz for GTX 285 GPU and about 125 fps for GTX 660 Ti GPU.
Table 5.4: Test Conditions

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset Used</td>
<td>TUM Night Dataset</td>
<td></td>
</tr>
<tr>
<td>ROI</td>
<td>$108 \times 448$</td>
<td>Dimensions (rows x columns) of the region of interest.</td>
</tr>
<tr>
<td>NUM_FRAMES</td>
<td>899</td>
<td>Total frames used to obtain average frame rate along the vertical axis of Figure 5.18.</td>
</tr>
<tr>
<td>NUM_GOOD_LINES</td>
<td>150, 300, 600, 1200, 2400, 4800</td>
<td>No. of particles used by Particle Filter (PF) for lane tracking. This is the variable that denotes the horizontal axis in Figure 5.18.</td>
</tr>
</tbody>
</table>

FPGA Resource Utilization

In Table 4.3, some of the key specifications of resources available on Altera Stratix-V FPGA are presented. While creating the hardware image file (*.aocx file) for FPGA, Altera Offline Compiler (AOC) determines what percentage of these resources are required to perform the computations in a given kernel. Resource utilization can also be manually affected using attributes and pragmas provided by Altera’s implementation of OpenCL.

Listing 5.2 presents an example to show how the kernel attributes are specified for FPGA kernels. This particular example is similar to Listing 5.1 presented before except the lines to mention kernel attributes are added for the case TUM Night Dataset. Using the attributes

```c
_attribute_((reqd_work_group_size(250,1,1)))
_kernel void createLineKernel(...)```

and

```c
_attribute_((reqd_work_group_size(50,1,1)))
_kernel void goodLineWeightKernel(...)```

respectively for kernel 1 and kernel 2, the required work-group size is chosen while executing the NDRange of each kernel. While compiling this particular kernel file to obtain the hardware design image file (*.aocx file) for the FPGA, the resource utilization is displayed by Altera Offline Compiler (AOC) which is presented in Table 5.5.
Listing 5.2: OpenCL Kernel File for FPGA

```c
#include "kernel_header.h"

typedef struct {
    float x[108];
    float y[108];
    float dist;
    float theta;
} lineStruct;

// ------------
// KERNEL 1
// ------------

__attribute__((reqd_work_group_size(250,1,1)))
__kernel void createLineKernel(
    __constant const struct createLineParams * restrict cline_params,
    __global const int * restrict img_grad,
    __global struct lineStruct * restrict line_output)
{
    size_t idx_dim1 = get_global_id(0);
    ...
    ...
}

// ------------
// KERNEL 2
// ------------

__attribute__((reqd_work_group_size(50,1,1)))
__kernel void goodLineWeightKernel(
    __global const lineStruct * restrict good_line,
    __global const lineStruct * restrict best_line,
    __global const int * restrict img_grad,
    __constant noiseStruct * restrict allnoises,
    __constant wtKernParams * restrict params,
    __global lineStruct * restrict good_line_out,
    ...
```
Chapter 5. Results

```c
__global float * restrict w_pf_out)
{
    size_t idx_dim1 = get_global_id(0);
    ...
    ...
}
```

Table 5.5: FPGA Resource Utilization for Executing Kernels for TUM Night Dataset

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Utilization</td>
<td>96%</td>
</tr>
<tr>
<td>Dedicated Logic Registers</td>
<td>51%</td>
</tr>
<tr>
<td>Memory</td>
<td>79%</td>
</tr>
</tbody>
</table>

Using the percentage values of various resource utilization as presented in Table 5.5, one can deduce how many elements of each of the resources mentioned in Table 4.3 are used for a particular FPGA design. In this case, about 96% of Logic Elements present on the FPGA board are used. This indicates that FPGA is running out of these elements and hence any increase in the computations within the kernels that use more of Logic Elements can make the design too big to run on this particular FPGA. For example, when the attributes

```c
__attribute__((num_compute_units(2)))
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(512,1,1)))
_kernel void createLineKernel(...)
```

and

```c
__attribute__((reqd_work_group_size(50,1,1)))
_kernel void goodLineWeightKernel(...)
```

are used, the corresponding resource utilization shown by AOC is presented in Table 5.6.

Value higher that 100% in Table 5.6 indicate that this particular design cannot be fit on the available FPGA. One workaround for this is to redesign the kernels such that less number of Logic Elements and Memory is used. Other,
Table 5.6: FPGA Resource Utilization for Executing Kernels for TUM Night Dataset

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Utilization</td>
<td>132%</td>
</tr>
<tr>
<td>Dedicated Logic Registers</td>
<td>66%</td>
</tr>
<tr>
<td>Memory</td>
<td>104%</td>
</tr>
</tbody>
</table>

An expensive solution would be to use more FPGAs than just one, to achieve this design. Using a higher number of Compute Units leads to a higher traffic over the PCIe host-device interface which would in turn slow-down that data transfers while at the same time almost doubling the number of computations that can be performed by doubling the Compute Unit attribute. Hence, one has to strike a balance between PCIe traffic generated and the number of computations performed. More information of kernel attributes and pragmas provided by Altera for its FPGAs can be found in the manuals provided by Altera (Altera (2013a), Altera (2013c) and Altera (2013b)).
Chapter 6

Conclusion

A lane detection and tracking algorithm is developed and tested on two different GPUs and an FPGA, using the same code base that was written using a combination of C/C++ and OpenCL. Lane Detection is done using a novel approach by first finding the gradients in the edge-detected image and assigning weights to particles that represent line markings, based on their distances to these edges. After lane detection, a Particle Filter is used for Lane Tracking, to enhance the computation efficiency of the algorithm. Creation of lines in lane detection step and prediction & measurement update steps of particle filter were written as OpenCL kernels so as to execute these parts of the algorithm on high-performance devices (GPUs and FPGA). The algorithm developed works quite robustly in different road scenarios as proved by testing it on five different datasets of varying light conditions, lane widths and road traffic as presented in chapter 5.

Significant variations in the performance were observed for the three high-performance devices. While NVIDIA GeForce GTX 660 Ti GPU has processed up to 104 frames/sec, NVIDIA GeForce GTX 285 GPU followed next with around 79 Hz. Altera Stratix V FPGA gave up to 27 Hz of frame rate. However, it is still a good performance for an FPGA that is designed using the C based OpenCL language. One could definitely derive significant performance increase for FPGA if it designed in low level languages like VHDL. But this would take a lot of expenditure in terms of time involved and difficulty in developing the design. Hence, the portability is the trade-off that OpenCL offers at the cost of a slight reduction in performance. Considering the higher value of floating point operations per second (FLOPs) per every watt consumed, offered by FPGAs and GPUs, as compared to the traditional ECUs (Electronic Control Unit), OpenCL has definitely opened
new doors to programming energy efficient and high performance hardware with ease. The results were quite satisfactory as the aim of the current work was to demonstrate the potential of programming heterogeneous high-performance hardware using same OpenCL code base for lane detection and tracking algorithm.

**Future Work**  Using the same OpenCL code, possibility of running this algorithm on a combination of FPGA, GPU and Multi-core CPU can also be explored which would deliver a much higher throughput, hence allowing other Driver Assistance Systems (DAS) to also run simultaneously along with the lane detection and tracking code on the future autonomous vehicles. This algorithm can also benefit from other inertial sensors in obtaining ground truth information to evaluate our simulation model. Also, using the camera calibration data to correct the image for perspective effect by applying Inverse Perspective Mapping (IPM) would help in detecting the multiple lanes on the road without having to limit to detecting and tracking.

Currently Altera’s implementation of OpenCL (version 1.1) for its FPGAs only comes with the support for *OpenCL Buffer objects*. This means that images to be used for computation in an OpenCL kernel have to be converted into arrays before sending them from host CPU to the FPGA. However, the later versions of OpenCL (version 2.0) come with the support to send the images in the form of *OpenCL Image objects* instead of buffers. This increases the ease of computations in the kernel. Also, memory access is faster from an image object as compared to very long arrays or buffers. Hence, if Altera comes with support for such later versions of OpenCL, there can be significant improvement in the performance of an FPGA.
Appendix A

Lane Markings at Intersections

Figure A.1: Road markings at signalized intersections (MUTCD 2013).
Bibliography

Printed Sources


Zhao, K., M. Meuter, C. Nunn, D. Muller, S. Muller-Schneiders, and J. Pauli
(2012). “A novel multi-lane detection and tracking system”. In: Intelligent Vehicles
Symposium (IV), 2012 IEEE, pp. 1084–1089 (cit. on p. 44).

**On-line Sources**


Craig Davis (2013). *Developing High-Performance Low-Power Solutions using FP-


edu/malaa/datasets/caltech-lanes/ (cit. on p. 73).

