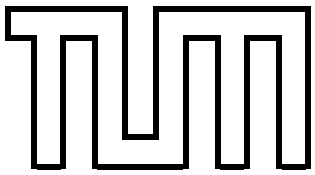
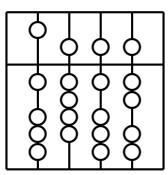


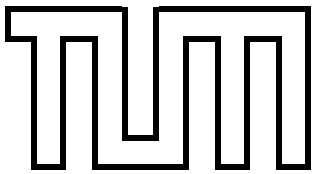
Department of Informatics
Technische Universität München

Bachelor's Thesis in Informatics

Real-time lane detection and tracking on high performance computing devices

Jan Botsch





Technische Universität München
Department of Informatics

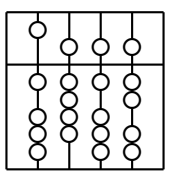
Bachelor's Thesis in Informatics

Real-time lane detection and tracking on high performance computing devices

Echtzeit Fahrspurerkennung und -verfolgung mit FPGAs und GPUs

Jan Botsch

Supervisor: Kai Huang, Dr.
Adviser: Biao Hu, M.Sc
Abgabedatum: 20. März 2015



Ich versichere, dass ich diese Bachelorarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

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

Munich, 20.03.2015 _____

Jan Botsch

Abstract

Road lane detection and tracking methods are the state of the art in present driver assistance systems. However, lane detection methods that exploit the parallel processing capabilities of heterogeneous high performance computing devices such as FPGAs (or GPUs), a technology that potentially will replace ECUs in a coming generation of cars, are a rare subject of interest. In this thesis a road lane detection and tracking algorithm is developed and implemented, especially designed to incorporate one or many, and even heterogeneous, hardware accelerators. Road lane markings are detected and tracked with a Sequential Monte Carlo (SQR) method. Lane detection is done by populating a pre-processed gradient image with randomly sampled, straight lines. Each line is assigned a weight according to its position and the best positioned lines are used to represent the lane markings. Subsequently, lane tracking is performed with the help of a particle filter. The code was tested on three devices, one GPU - the *NVIDIA GeForce GTX 660 TI* - and two FPGAs - the *ALTERA Stratix V* and the *ALTERA Cyclone V SOC*. The tests revealed a processing frame rate of up to 627 Hz on the GPU, 478 Hz on the Stratix V FPGA and 38 Hz on the Cyclone V SOC. They also showed a significant improvement in accuracy and robustness, a 2.4-4.6 times faster execution on the GPU, a 8.4-29.7 times faster execution on the Stratix V and a reduction of memory consumption by 71.94 % compared to a similar lane detection method. The algorithm was tested on different recorded videos, on independent benchmark datasets and in multiple test drives, confronting it with a wide range of scenarios, such as varying lighting conditions, presence of disturbing shadows or light beams and varying traffic densities. In all these scenarios the algorithm proved to be very robust to detect and track one or multiple lane markings.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

Road accidents lead to thousands of fatalities in Germany every year (destatis, 2013, p. 7). The vast majority of these accidents are caused by driving errors, e.g. 86 % in 2012 (destatis, 2013, p. 15), and many can be avoided by driver assistance systems (DAS). A variety of DAS exist that aim to improve road and driver safety. One of these system is the lane departure warning system that uses a road lane detection algorithm to detect if a car departs from a lane. Today lane departure warning systems only notify the driver and do not intervene, but their role will become more active in the future as the automation of driving progresses.

The increasing responsibility poses new challenges to DAS. They have to be more reliable, more robust and react even faster. Consequently it will be demanded of lane departure warning systems that they detect a departure from a lane fast enough to be able to counteract it. In order to do so, road lane detection algorithms are necessary that deliver very accurate and fast estimates of lane positions.

Road lane detection algorithms are already the state of the art and exist in many variations. Most algorithms are vision-based and process a stream of frames depicting the road. The required computations are performed on electronic control units (ECUs), application-specific embedded systems. Tens or even up to a hundred ECUs are integrated in modern vehicles, which has resulted in an enormous complexity and power consumption. However, recent developments have made other processing units such as GPUs and, more importantly, FPGAs a feasible alternative or complement to ECUs. Especially FPGAs promise faster and more flexible processing at less power consumption. This development has progressed as far as the manufacturer Xilinx has already started to design automotive ECUs that embed FPGAs (F. Fons and Fons, 2012).

The introduction of new processing units - mostly hardware accelerators - to the automotive market transforms the demands on the software. Methods and algorithms have to be adapted to the physical layout of the computing device in order to achieve a maximal computation speed.

Existing lane detection methods, however, are designed for conventional ECUs. Though some methods may be adapted to the parallel computation structure of GPUs and FPGAs, they were not specifically developed for this purpose and hence suffer from bottle necks and other disadvantages. And they certainly lag behind a method that is specifically designed for the use of hardware accelerators.

1.2 Proposed Method

In this paper a novel road lane detection and tracking method is developed that is specifically tailored for heterogeneous hardware accelerators. The work started off as an extension to an previously developed method published in (Madduri, 2014). Though promising, this work quickly proved insufficient, as the detection is neither accurate nor fast enough for a realistic application. Further investigation revealed that this stems from several design and implementation errors. Therefore this thesis took up the task to develop a method that follows a similar structure as proposed in (Madduri, 2014), but differs significantly in the design and implementation of the lane detection and tracking methods.

The proposed method is vision-based and aims at detecting the lane markings, as their appearance significantly divers from the underlying road. It requires no knowledge of any physical parameters like the position and orientation of the camera and is hence very flexible.

1.2.1 Structure of the method

In a first step, the algorithm pre-processes the incoming frames to emphasize relevant information. Then, the algorithm may take two different ways. Either a random sampling method is applied that samples hundreds of possible lane markings, weights them according to their accuracy and selects the best estimates. Or, if previous estimates of the lane marking positions are available, a computationally less expensive particle filter is used to track the lane markings. The latter is applied in the vast majority of the frames.

Both random sampling for lane detection and the particle filter for lane tracking proved so efficient that additional steps, like correcting the perspective with inverse perspective mapping (IPM), are simply unnecessary.

The computation in all three stages, pre-processing, lane detection and lane tracking, can be partly or completely performed in parallel. The respective parts were implemented in the form of OpenCL kernels and are executed on a hardware accelerator.

1.2.2 Verification of the performance

The developed methods were implemented as an algorithm and extensively tested. This includes tests with recorded videos, with independent datasets and in test drives. The testing was carried out on three different hardware accelerators:

- Nvidia GeForce GTX 660 TI GPU
- Altera Stratix V FPGA
- Altera Cyclone V FPGA

1.3 Structure of this paper

The next chapter, chapter 2, provides comprehensive information on the tools that are used in this thesis for developing and implementing the lane detection and tracking method. This includes a theoretical background of the particle filters as well as specifics on the OpenCL framework that is used for the implementation.

In chapter 3 the lane detection and tracking method of this thesis is explained in detail. This is complemented by Chapter 4 that describes specifics on the implementation and extensions to the algorithm. Chapter 5 presents and discusses the results of the testing and chapter 6 summarizes the thesis.

2 Background

2.1 Particle Filter

Particle filters belong to the family of Sequential Monte Carlo methods (SCM) and use a set of particles to describe a posterior density distribution of a state-space model. They require little or no assumptions on the model, only periodic (indirect) measurements of the true state. Indirect position "measurements" in the form of images are the only available information in a vision-based lane tracking algorithm, making a particle filter a promising tool for lane tracking.

Particle filters implement the Bayesian recursion equation

$$P(X|Y) = \frac{P(Y|X)\,P(X)}{P(Y)} \tag{2.1}$$

in order to determine the posterior density distribution $P(X|Y)$ of a state X given the measurements Y (Gordon, Salmond, and Smith, 1993). $P(X)$ describes the distribution prior to taking measurements, $P(Y|X)$ the likelihood of the measurement conditioned on the state X and $P(Y)$ the evidence, or the overall probability of the measurement.

Equation (2.1) is implemented by sampling a number of weighted particles $i = 0, 1, \ldots N$ from the prior distribution. Each particle has a state vector X_i transforming equation (2.1) to

$$P(X_i|Y) = \frac{P(Y|X_i)\,P(X_i)}{P(Y)} = \frac{P(Y|X_i)\,P(X_i)}{\sum_{i=0}^{N} P(Y|X_i)\,P(X_i)} \tag{2.2}$$

In addition, each particle is assigned an importance weight w_i. The importance weight expresses the likelihood that X_i is identical with the true state of the state-space model given the measurement Y, hence $w_i = P(Y|X_i)\,P(X_i)$. Inserting this in formula (2.2) yields

$$P(X_i|Y) = \frac{P(Y|X_i)\,P(X_i)}{\sum_{i=0}^{N} P(Y|X_i)\,P(X_i)} = \frac{w_i}{\sum_{i=0}^{N} w_i} \tag{2.3}$$

In this work equation a particle filter is implemented in three consecutive steps that are performed for each frame:

Prediction update The state X_i of each particle is updated in the same manner as the true state is expected to change.

Importance weight update The importance weight w_i is calculated for each particle and formula (2.3) is evaluated.

Resampling A resampling step is introduced that randomly samples particles according to their importance weight and copies them to a new, equally sized set. Particles may be copied more than once. This yields a set with a higher density of "good" particles and avoids a degeneration of the particle set (Cappe, Godsill, and Moulines, 2007, pp. 3-4).

A detailed description of the particle filter for the lane tracking based on formulas 2.1-2.3 is presented in section 3.4.

2.2 High performance computing with hardware accelerators

The idea of using hardware accelerators to support central processing units (CPUs) in computationally expensive operations has been around since the early times of computers. The first personal computers, such as the IBM PC, included hardware accelerators specifically designed to speed up floating-point calculation.

Since then, the role of hardware accelerators has changed significantly. The rapid increase in computing power of CPUs has made many hardware accelerators obsolete. In fact, till the early 2000s, the field of high performance computing (HPC) focused on increasing the clocking frequency of single-core CPUs in order to meet the growing demand for computing power. The dynamic power consumption of a CPU can be described by the relation

$$\text{POWER CONSUMPTION} = \text{CAPCITY} * (\text{VOLTAGE})^2 * \text{FREQUENCY} \tag{2.4}$$

The minimum voltage is dependent on the frequency, so that if the frequency is risen the power consumptions rises in the scale of $\text{FREQUENCY}^{(2-3)}$.

This and the rising awareness that many applications are better suited to thread level parallelism (TLP) methods, led to the development of multi-core CPU's, consisting first of two and today already of up to ten cores. They exert a better performance and a reduced power consumption, as each individual core operates at a lower frequency than the single cores that were used before.

Still, there are numerous applications where even multi-core CPUs proved inefficient (Navarro, Hitschfeld-Kahler, and Mateu, 2013). This is the case for applications which would allow extensive parallel computing, like the rendering of 2D- and 3D-images for graphical interfaces, where the pixels are (mostly) independent and can be processed simultaneously.

2.2.1 Graphical Computing Units (GPUs)

The development of a new hardware accelerator, the Graphical Computing Unit (GPU), addressed this issue. As the name suggests, GPUs were specifically developed to perform graphical calculations, which reflected on their hardware layout. As of 2006 most GPUs were designed as a so-called graphic pipeline. The input were geometric primitives which passed through several stages: vertex operations, primitive assembly, rasterization, fragment operations, and composition to yield a final image (J. Owens et al., 2008, p. 880; J. D. Owens et al., 2007, p. 82). Each element of this pipeline was realized by a separate hardware unit and multiple copies of the pipeline were printed on a GPU. This allowed the simultaneous execution of thousands of operations to produce images on the screen.

The advantage of GPUs over CPUs in graphical applications is obvious. However, it was realized that there are many other applications, such as the simulation of molecules in chemistry or the Navier Stokes equation in engineering, which display a potential for using a highly parallel computation structure. This led to changes in the hardware layout of GPUs, giving rise to general purpose GPUs (GPGPUs).

GPGPUs display a more generalized hardware, which makes them suitable for a wide range of algorithms. As of today they have developed into an equivalent to CPUs in the field of parallel computing. They are used increasingly in (scientific) supercomputers and other high performance applications (Navarro et al., 2013). And their success is not limited to that. As applications in general require more and more computation power, GPUs are increasingly implemented in mobile phones, tablets and other low cost devices (Peddie, 2013).

The manufacturer Nvidia announced in 2008 that it would aim to sell GPUs for applications in automotive applications (Muyan-Özçelik and Glavtchev, 2008) and at least since 2010/2011 GPUs are used in vehicles produced by BMW and AUDI (Cunningham, 2011). Though these applications

mostly serve in entertainment systems, it can be expected that in the future they will be used for DAS like the lane departure warning system.

2.2.2 Field-Programmable Gate Arrays (FPGAs)

Another hardware accelerator that can be used for parallel computing is a field- programmable gate array (FPGA). FPGAs are highly configurable devices with a set of hardware resources (logic blocks, routing channels and I/O Pads) that can be combined in different ways to perform different functionalities or programs. Unlike CPUs (or GPUs), where functionality and routing of hardware blocks is irreversible after the manufacturing stage, FPGAs can change their hardware layout at run time. Desired layouts are stored on a non-volatile flash memory and can be loaded whenever it is necessary.

Programmes that are written for a CPU are compiled to fit the hardware layout. In FPGAs, however, the hardware is routed to fit a programme's requirements. This allows a very efficient and dynamic combination of both parallel and serial computation parts.

This flexibility used to come at the cost of a significantly higher power consumption than the CPU/GPU counterparts. However, a number of architectural innovations, among them 28G transceivers and partial reconfiguration, have increased the efficiency of FPGAs in many application to or even beyond the level of GPUs and CPUs (Tian and Benkrid, 2010). In (Yazbeck and Kenny, 2012) the manufacturer Altera claims, for example, that it was able to reduce the power consumption of its Stratix FPGAs by 80 percent between 2004 and 2011, while increasing the I/O bandwidth drastically.

As FPGAs promise an efficient computation in a wide range of tasks, they have become a standard in many industries. The steady decrease in power consumption has made them a real alternative to existing technologies in markets, such as the automotive industry, where power consumption is a huge concern. Here FPGAs have a high potential to replace or complement the currently used ECUs in vehicles, since they offer far more flexibility. As (F. Fons and Fons, 2012) shows, first attempts in this direction have already been made.

With this development in mind, the lane detection and tracking method of this thesis was tailored for the integration of heterogeneous hardware accelerators, such as FPGAs and GPUS.

2.2.3 OpenCL

The change to a more general hardware layout of GPUs, gave rise to the development of new, equally general APIs replacing the old graphic APIs. Developers like ATI (now AMD Graphics Product Group) and Nvidia created in 2006/2007 Close to Metal (CTM) and Compute Unified Device Architecture (CUDA), parallel computing platforms that supported their respective GPGPU architectures. The wish for a unified programming model targeting heterogeneous platforms led to the development of OpenCL 1.0 in 2009. OpenCL is maintained by the non-profit technology consortium Khronos Group and it quickly evolved into versions 1.1, 1.2, 2.0 and 2.1 adding a wider support for devices and functionalities. The OpenCL specification provides the following four models to organize the execution of programmes embedding hardware accelerators:

Platform model This model defines one processor that coordinates the execution (host) and one or more processors capable of executing C-like OpenCL code (devices). It further specifies that the code for the devices is organized in C-like OpenCL functions (called kernels) that execute on the devices.

Execution model Outlines the OpenCL environment, the frameworks different abstraction levels and its use in a C/C++ program, including the setting up of an OpenCL context, the organisation of host-device interaction and the different concurrency models for the kernel executions (e.g.

SIMD). It specifies that a C/C++ programme is executed on the host and the host distributes tasks between the devices, which execute the required kernels. The kernels are executed in workgroups consisting of one ore many work items. Within a work group each work item executes the same kernel, but different work groups may execute different kernels.

Memory model Defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture. Four different memory types are provided,

- global memory: is very big, all processing elements can access it, but access has a high latency.
- constant memory: can be written to only by the host. It is smaller than the global memory and has a lower latency.
- local memory: is shared by a workgroup (of processing elements), smaller than constant memory, but access is faster.
- private memory: belongs to a particular work item, mostly implemented by registers. It is the fastest and smallest memory.

Programming model specifies the mapping of the concurrency model to physical hardware.

If OpenCL is used with multiple devices from different vendors, an OpenCL implementation for each device has to be available on the target platform (the host). In addition an installable client driver loader (ICD loader) is required for each implementation, in order to use different OpenCL implementations in the same application.

2.2.4 Setup for testing

The lane detection and tracking algorithm developed in this thesis is tested on two different hosts and three different devices. First, an Intel(R) Xeon(R) Processor E3-1225 (6M Cache, 3.10 GHz) host with 20 GB RAM is used. It is combined with two devices, the Altera Stratix V FPGA and the Nvidia GeForce GTX 660 TI GPU. The latter has a higher computing power and cane therefore be expected to deliver a better performance. The algorithm is executed and evaluated using this host with each device separately and with both devices together.

As mentioned, an OpenCL implementation and ICD loader from Nvidia and Altera are required to use the GPU and FPGA in the same application. Altera's OpenCL (AOCL) framework was used in the version 13.0sp1, which bases on OpenCL 1.0 and does not yet provide an ICD loader. Therefore a custom ICD loader for the AOCL was developed and is outlines in section 4.2.

Second, a Cyclone V System on a Chip (SOC) is used, consisting of an ARM Cortex-A9 processor (1GB RAM) as host and a Cyclone V SX FPGA (1GB RAM) as device. The Cyclone SOC is a cheaper and less powerful system than the first system and its performance cannot compete. It was used, because it is compact and allowed live testing of the algorithm in a vehicle, which will be presented in section 5.4.

3 Method

The lane detection and tracking method presented in this chapter processes an incoming live video stream frame by frame and extracts the position of lane markings. The video stream, showing a road and the area surrounding it, will be delivered by a camera installed in a vehicle or mobile phone. A frame is processed in two subsequent steps. First, information on the lane markings' position is amplified and extracted from the frame in a *pre-processing* stage. Then, depending on whether previous estimates of the position exist or not, the exact position of the lane markings is detected in a *lane detection* step or tracked in a *lane tracking* step. This process is illustrated in figure 3.1 and will be explained in the following in detail.

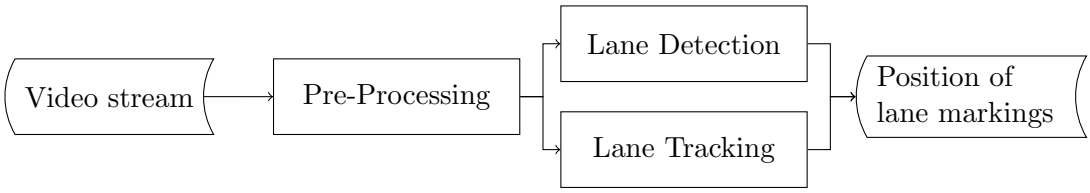


Figure 3.1: *The flow of the lane detection and tracking method presented in this thesis.*

3.1 Pre-Processing

In the pre-processing stage information on the position of lane markings is extracted from a frame and passed on to the lane detection or tracking step. The pre-processing stage itself consists of four different procedures that are applied successively to the raw images. The procedures are

1. **ROI selection.** A region of interest (ROI) is defined within the raw image and only this region is further processed.
2. **Grayscaling**. The ROI is transformed to a grayscale space
3. **Edge detection.** A Sobel operator detects edges in the image
4. **Thresholding.** Disturbances are eliminated with the help of threshold values.

The flow in the pre-processing stage is shown in figure 3.2.

3.1.1 Region of Interest

Figure 3.3a shows an image taken from within a car. Only a small part of the image is of actual interest, the part that shows the street. This part, highlighted in green in figure 3.3b, will henceforth be called the region of interest (ROI). The ROI contains all essential information for the subsequent lane detection step and the rest of the frame can be discarded. The size of the ROI is a driving parameter that determines computational speed and effort of the lane detection/tracking. The smaller the ROI is chosen, the faster the lane detection performs.

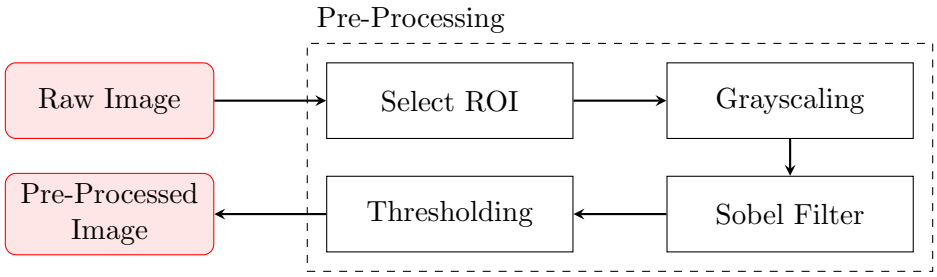


Figure 3.2: *Flow and elements of the pre-processing stage*

In this work size and position of the ROI are kept adjustable, so that the algorithm can be tested in a wide range of a scenarios. In a real application with a fixed camera both size and position can be hard-coded, which will yield an additional acceleration of the computation. Once the ROI is selected, only the area within the region is processed in the subsequent stages.

3.1.2 Grayscaling

A raw image from a camera (and hence the ROI) is provided in the RGB colour format. In this format each pixel is assigned three colour channels, one for red, one for green and one for blue. The values of the three channels are combined to yield the actual colour of the pixel.

In the RGB colour format the distinction of lane markings from their environment is challenging or not possible. First, all three colour channels would have to be compared. Second and more important, the comparison would have to be generic to account for varying colours (e.g. white or orange) and intensities of lane markings.

It is more promising to make use of the characteristic that lane markings are substantially brighter than the road they are printed on. The ROI can be transformed to a grayscale format, where each pixel reflects the intensity of the pixel in the original image. Dark pixel will receive low intensity values and bright pixels will receive high values.

Different methods exist to transform an image from the RGB to the grayscale format, all based on a weighted summation of the different channels. In this thesis a resource saving numerical approximation is used for the transformation. It avoids the use of floating point arithmetic, which accelerates the computation on hardware accelerators, especially on FPGAs. The transformation formulas are

$$Y = 66 \cdot R + 129 \cdot G + 25 \cdot B \qquad \text{(conversion to gray scale)} \tag{3.1}$$

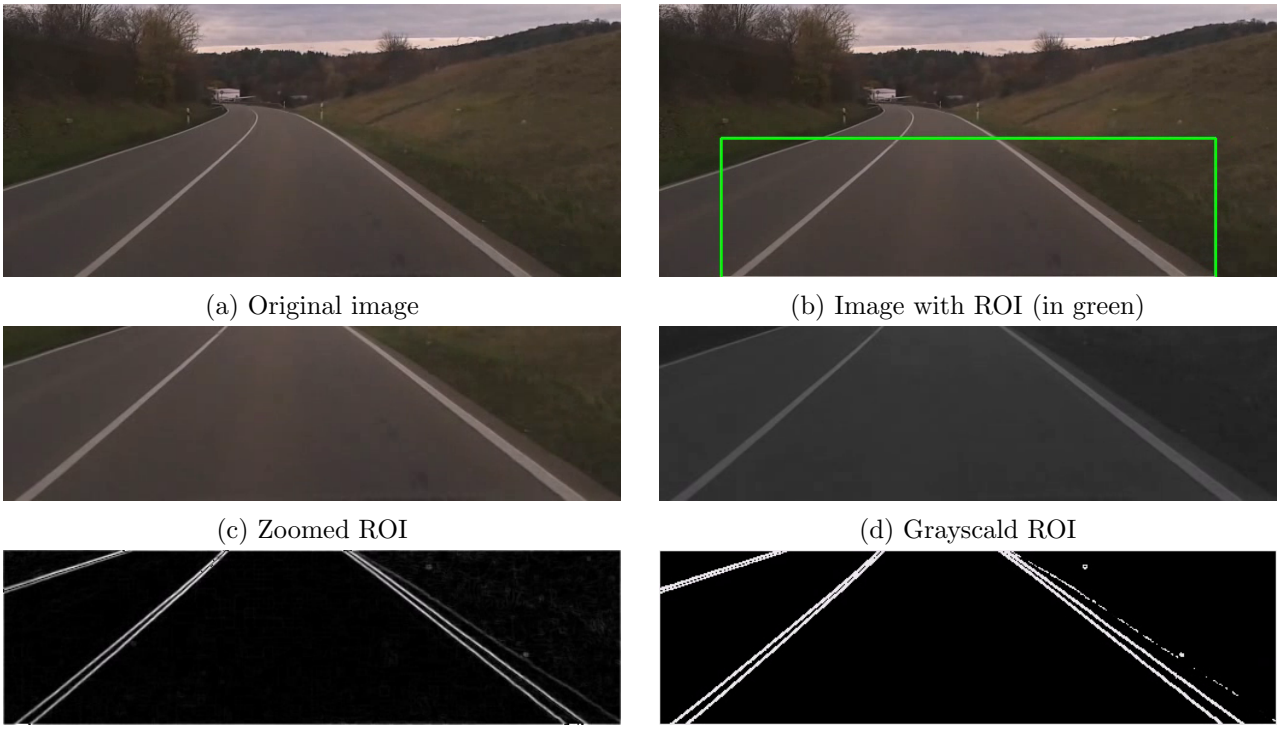
$$Y = (Y + 128) \gg 8 \qquad \text{(downscaling to 8-Bit with rounding)} \tag{3.2}$$

$$Y = Y + 16 \qquad \text{(shifting of values)} \tag{3.3}$$

where R is the red colour channel of the pixel, G the green and B of the blue, all in the range $[0, 255]$. Y is the intensity and will have a value in the range $[16, 235]$. This conversion is performed for each pixel of the ROI individually, resulting in a grayscaled image as shown in figure 3.3d.

3.1.3 Edge detection

A Sobel filter is applied to the grayscaled image producing a new image, where only transitions and edges (such as the lane markings) of the original image are present. More technically, each pixel in the new image describes the gradient of the original image at that position. A strong gradient will be represented by a high absolute value and a soft gradient by a value close to zero. The gradient is



(a) Original image

(b) Image with ROI (in green)

(c) Zoomed ROI

(d) Grayscald ROI

(e) Output of the Sobel filter

(f) Final output after thresholding

Figure 3.3: *Original image and output of the different pre-processing stages.* In (a) the original image depicting a road is shown. In (b) a ROI is marked that contains all required information for detecting/tracking lane markings. (c) shows a zoomed image of the ROI and (d) the result after grayscaling. In (e) the lane markings and other edges were detected by a Sobel filter. In (f) the lane markings were highlighted and disturbances were removed by using a threshold-value

calculated by applying two discrete differentiation operators to the image, one for the horizontal and on for the vertical direction. The operators include smoothing and have the form of two 3x3 kernels, which are convoluted with the original image. The formulas for the convolutions are

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \text{Image}, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \text{Image} \tag{3.4}$$

The formulas show that 9 pixels are required in order to determine the gradients at a position. G_x and G_y are then combined to yield the overall gradient. Several methods exist to combine the gradients, here a computationally inexpensive variant, shown in equation 3.5, is used.

$$G = |G_x| + |G_y| \tag{3.5}$$

Figure 3.3e illustrates the output of the Sobel filter. The Sobel filter delivers optimal results in an environment consisting of a dark street, bright lane markings and no disturbances. Here the lane markings are represented by pixels with high intensity and the rest of the image is populated with zero intensity pixels.

3.1.4 Thresholding

In a real application, however, images contain noise. This noise will be present in the form of additional, undesired edges. Possible sources for edges are:

- Other road markings (e.g. arrows for turns)
- Signal posts at the side of a road
- Varying colours of the street material
- Shadows and many more

At night, additional edges will be produced by street lights, light beams of other cars and reflections.

Some disturbances will only create thin and weak edges. These disturbances can be deleted from the image by thresholding: The intensity of all pixels, whose gradient falls below a certain threshold, are set to zero. Since the lane markings receive high gradients by the Sobel filter, they will not be affected negatively.

On the contrary, all pixels that have an intensity above the threshold are set to a maximum value. This further emphasises strong edges like the ones created by lane markings and eases the task of the subsequent lane detection/tacking step.

More importantly, it will make the lane detection possible under difficult conditions, e.g. when rain or fog lead to blurred images. In those situations the edges of the lane markings might be less visible but thresholding makes them brighter. The final output of the pre-processing stage after thresholding is shown in figure 3.3f. The thresholding formula that is applied on each gradient-pixel with intensity I, is:

$$I = \begin{cases} 0 & \text{if } I < \text{threshold} \\ \text{MAX\_VAL} & \text{if } I \geq \text{threshold} \end{cases} \tag{3.6}$$

3.1.5 Parallelization of the pre-processing

The pre-processing displays a high potential for parallel computations. Grayscaling and Thresholding can be applied to each pixel independently and in parallel. The use of the Sobel Filter requires knowledge about nine neighbouring pixels. This provided, all pixels in the ROI can be pre-processed in parallel. This yields an enormous speed-up of the computation, since there are tens of thousands pixels even in a small ROI.

An OpenCL kernel was developed that performs the pre-processing completely on a hardware accelerator (e.g. FPGA or GPU). Algorithm 1 shows the basic structure of the kernel.

3.1.6 Summary of the Pre-processing

The pre-processing is the first stage for detecting or tracking street lanes. A region of interest is selected from an incoming image. This region is pre-processed to provide the required information on the lane markings.

First the image is transformed to a grayscale space. Then a Sobel filter detects edges in the image and finally a thresholding step removes minor disturbances and strengthens the appearance of the lane markings.

The pre-processing is performed in parallel on a hardware accelerator (FPGA or GPU) in order to achieve maximal computation speed. The result is an image, where the lane markings (and some remaining disturbances) will be characterised by bands of pixels with high intensity.

Algorithm 1 Pre-processing

```
Load nine required PIXELS
for all PIXELS P do
    Transform P to grayscale space      (Grayscaling)
    Calculate G_xp and G_yp      (Sobel filter)
    G_x += G_xp
    G_y += G_yp
end for
Calculate G = |G_x| + |G_y|
if G < thres then        (Thresholding)
    G ← 0
else if G ≥ thres then
    G ← MAX.
end if
```

In the following sections on*lane detection* and *lane tracking*, the pre-processed ROI will be used to determine the exact position of the lane markings. Before that, the next section will introduce an assumption on the shape of lane markings and establish an efficient way to describe their position.

3.2 Description of lane markings

In a pre-processed ROI each lane marking will be represented by a continuous band of pixels with high intensity (see figure 3.3f). Eventually, information on the position and shape of this band is required, which is then passed on to a subsequent DAS. Before the shape and position of a lane marking can be determined, a method to describe them in an algorithm is required. This description has to meet three demands, namely it should

1. be able to represent a lane marking as accurate as possible
2. be universal and easy to use in a subsequent DAS
3. use as less memory and computational resources as possible

Whereas demands one and two are obvious, the third requirement stems from the nature of the algorithms used in this thesis. Both the lane detection and lane tracking are implemented with the help of random sampling methods. This includes the creation of a lot of samples for lane markings, in the following called *candidate lines*, from whom the most fitting candidate line will be chosen (more in sections 3.3 and 3.4). As there might be several thousand candidate lines, each should consume as less resources as possible.

3.2.1 Straight-Line-Assumption

If a lane markings appears in a complicated shape, it is challenging to find a feasible description. Either a complex mathematical function, or a set that contains all the pixels of the lane marking would be required.

The description is greatly simplified by the observation that lane markings mostly take the shape of a straight line within the ROI (compare figure 3.3f). This is due to the fact that the ROI captures only a small section of the street ahead.

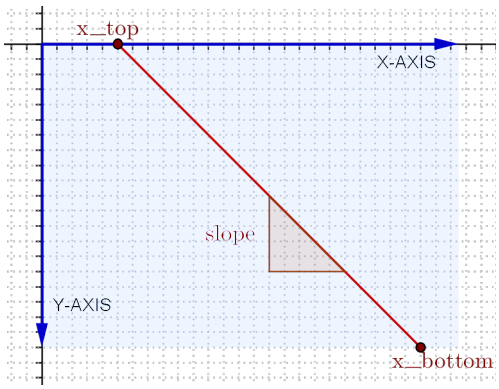


Figure 3.4: *Description of a lane marking.* x_{start} and x_{end} and the height of the ROI (the light blue area) fully define the line.

On straight roads and even in moderate bends the straight-line-assumption always holds. In sharp bends or other exceptionally routed roads, the lane markings might exhibit a bend. Then the ROI can be split horizontally into two or more regions, yielding subregions with straight lane markings.

3.2.2 Position description

Given the straight-line-assumption, the challenge of describing shape and position of a lane marking reduces to describing a mathematical line.

Previous Approach. Still, there a different ways of describing a line for the purpose of lane detection. In (Madduri, 2014) two arrays are used to store position and shape of a lane marking, one for all the x-values of the line and one for all the y-values. In addition, the angle of the line is stored and used to calculate all x- and y-values.

This approach is intuitive and thorough. Every bit of information on the lane markings is stored. It exerts, however, two disadvantages. First, the angle requires trigonometric functions for calculations. These functions are expensive on hardware accelerators (especially FPGAs) and consume a considerable amount of the available computing resources.

Second, each line is excessive in terms of its memory consumption. In a ROI of N rows, a single line will need storage of $2N+1$ values ($N \cdot \text{x-values} + N \cdot \text{y-values} + 1 \cdot \text{angle}$). Since thousands of candidate lines are used, the available memory on a hardware accelerator might not be sufficient.

New Method In this thesis the description of lane markings is simpler and more efficient. It bases on the characteristic that a straight line can be defined by two points on the line. The points at the top and the bottom of the ROI are chosen to represent a line. The position (or state) of a line can thus be expressed as

$$X = \begin{pmatrix} x_{\text{top}} \\ x_{\text{bottom}} \end{pmatrix}$$

if the height of the ROI is known. Since the y-values of the points are constant and given by the respective line number in the ROI (first line and last line), they do not need storing. Figure 3.4 shows a line in the ROI that is completely defined by the state X and the height of the ROI.

3.2.3 Comparison of Methods

The new approach has significant advantages over the method from (Madduri, 2014), which are:

Reduced memory consumption The memory a line consumes, was reduced from $2N + 1$ to exactly two values (compare figure 3.4).

Simple mathematical operations The need for trigonometric operations was eliminated. The slope s_X and any other point with y-value n can be determined by

$$s_X = \frac{(x_{\text{bottom}} - x_{\text{top}})}{\text{ROI\_HEIGHT}}, \qquad x_n = x_{\text{top}} + s_X \cdot n$$

Both formulas use only basic mathematical operations that are inexpensive on FPGAs.

3.3 Lane Detection

Given a frame, the positions of the lane markings have to be found. In the pre-processing stage a ROI was created, where the lane markings are indicated by a band of pixels with high intensities. It is the task of the lane detection or lane tracking algorithm (section 3.4) to extract the exact positions of the lane markings.

Lane detection is performed whenever no estimates on the lane markings are available. An example is the first the very first frame that is processed. The ROI is given, but we cannot rely on any information from past frames. After detecting the lane markings in the first frame, lane detection is only performed when the lane tracking algorithm fails to track the markings (more about that in section 3.5). This is done due to the high resource consumption of the lane detection algorithm.

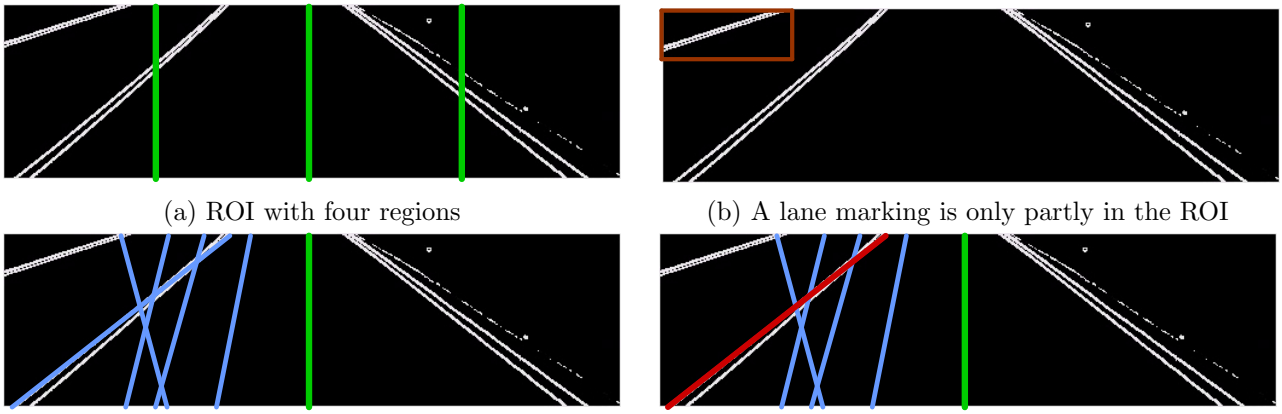
3.3.1 Characteristics of lane markings

There are three characteristics of lane markings that substantially reduce the effort to detect them. First, markings that define a lane do not cross and have a distance between them. If N lane markings have to be detected, they can be searched in N distinct regions. Figure 3.5a shows the division of the ROI into four regions, where lane markings are expected.

Second, lane markings do not necessarily lie completely within a region or even the ROI. It is a common situation that lane markings cross into or out of a region, e.g. when a car changes a lane. This is shown in figure 3.5b, where a lane marking is only partly contained in the ROI.

Third, lane markings are thicker than one pixel. They mostly have a width between 5 and 15 pixels, which has to be considered in the algorithm. Together, these characteristics give rise to the following basic structure of the lane detection method:

1. Divide the ROI into equally sized regions
2. Detect a lane marking in each region
    - Consider the width of the marking
    - Account for lines that cross regions



(a) ROI with four regions

(b) A lane marking is only partly in the ROI

(c) ROI with sampled lines in one region

(d) ROI with sampled and best line

Figure 3.5: *Detection of lane markings.* (a) shows the division of the ROI in four equally sized regions. (b) shows a lane marking that is only partly within the ROI. In (c) one subregion is filled with randomly sampled candidate lines and in (d) a candidate line that represents the lane marking is highlighted in red.

3.3.2 Lane Detection Method

But how can a lane marking be detected in a region? Here the lane detection algorithm makes use of the straight-line-assumption from section 3.2.1. Each region is populated with many, randomly placed candidate lines, as shown in figure 3.5c. It can be seen that some candidate lines match the lane marking better than others, since they were placed randomly.

The actual detection is achieved by assigning a "weight" to the candidate lines, which expresses how close the line is placed to a real lane marking. The weight of a line is determined by summing up the intensities of all pixels the line has in common with the ROI.

Further, also the pixels in an adjustable neighbourhood around a line (left and right of the actual line) are added to this weight. This accounts for the width of a lane marking.

Keeping in mind that lane markings are represented by pixels with high intensities, this method results in candidate lines with heigh weights, if they are close to a lane marking. The candidate line with the highest weight in a region is selected to represent the lane marking, as shown in figure 3.5d.

The number of candidate lines that are sampled in a region is adjustable and there is a trade-off between the performance of the lane detection and its quality. Using more candidate lines will in general result in a better detection, but also consumes more time.

3.3.3 Sampling of a Line

An individual candidate line is created by sampling its x_{top}- and x_{bottom}-values from a normal distribution. The two defining parameters of a normal distribution, the mean μ_{sam} and the standard deviation σ_{sam}, are set in a way that candidate lines in different orientations are sampled. This is important to detect lane markings with odd orientations and positions.

The mean of a normal distribution represents the value we expect to appear (the expectation). In the case of lane detection the most likely position of a lane marking is the centre of a region. This is the position of a lane marking, if a vehicle drives on a straight road.

The standard deviation decides how the candidates are distributed around the expectation. If σ_{sam} is selected very small, mostly lines close to the centre will be sampled, neglecting the outer parts of

the region. If it is selected very high, the lines are spread far and the actual lane marking might be missed.

In this paper σ_{sam} was set to region_width/2. From the definition of the standard deviation of a normal distribution follows that statistically about 68 % of the sampled lines will be placed completely within a region and the other 32 % of the lines might overlap to other regions. This choice of parameters proved to detect lane markings in all kind of positions, as the results in chapter 5 will show.

3.3.4 Parallelization of the Lane Detection

The presented method for detecting lane markings has the desirable characteristic that the candidate lines are independent from one another. The sampling of the lines and the calculation of their weights can be done in parallel. Hence, the algorithm performs these parts on the hardware accelerator. Algorithm 2 shows an OpenCL kernel in pseudo code that creates candidate lines and determines their weight.

Algorithm 2 Lane Detection

```
1: x_top = centreline_offset + sample_from_Gaussian
2: x_bottom = centreline_offset + sample_from_Gaussian
3: s_x = (x_bottom − x_top) · one_divided_by_row_numbers
4: for all ROWS R do
5:     x_current = x_top + R · s_x
6:     weight += intensities (x_current + neighbourhood)
7: end for
```

The selection of the best lines from the candidate lines requires knowledge of all lines and is therefore performed on the host.

The lane tracking algorithm, which will be described in the following section, requires a set of possible candidates/particles for each lane marking. Therefore not only the best candidate is stored, but for each lane marking a set with a few dozen candidate lines is kept. In the following this set will be referred to as the *good lines*. The line with the highest weight amongst the good lines will be called the *best line* and represents the actual lane marking.

3.4 Lane Tracking

As the name suggests, lane tracking performs the task of tracking lane markings in a given frame. The output are exact positions of the lane markings, like in the lane detection step. Lane tracking differs from lane detection insofar as it uses information from a previous fame to detect the markings in a subsequent frame. Hence it does not actually detect the lines, but rather tracks them. It is designed as a very fast and inexpensive algorithm that is used whenever information from a previous frame are available.

The lane tracking algorithm has two sources of information at its disposal, the pre-processed ROI and the set of good lines and best lines from the previous frame.

A particle filter is employed to keep track of the markings. Each lane marking is tracked separately, which means that multiple instances of the same particle filter are used in the algorithm. The theory of particle filters was provided in section 2.1. In this section the particular implementation of a particle filter for the purpose of lane tracking is discussed.

3.4.1 Setup of the particle filter

The particle filter bases on formula 2.1 from section 2.1. Before this formula can be evaluated, its basic elements, the state variable X, the prior probability distribution $P(X)$ and the state observation Y have to be matched to the lane detection application. The state variable is easily defined as the position of a lane marking: $X = (x_{\text{top}}, x_{\text{bottom}})$.

Deriving a prior distribution $P(X)$ is more of a challenge, because the probability distribution of lane markings in a given ROI is not universally expressible by an analytical function. However, the prior distribution can be simulated by an empirical set of particles. Such a set is available for each frame in form of the *good lines* from the previous frame.

In a similar fashion the observations Y of the actual lane marking positions can be obtained. We cannot obtain direct observations, as there is no sensor that measures the real positions of the lane markings. But the *best lines* from a previous frame can be interpreted as observations of the lane markings in the current frame, assuming that the positions of the markings do not change significantly between two subsequent frames.

3.4.2 The lane tracking algorithm

Having determined the setup of the particle filter, the lane tracking algorithm can be implemented. It follows the outline of algorithm 2 for the lane detection with some minor adjustments. The individual steps were presented in section 2.1.

Prediction update. Step one from algorithm 2, the sampling of candidate lines from a normal distribution, can be omitted; the particles are already given in form of the good lines. Instead, an additional prediction update step is introduced.

This step is required, because the particles (*good lines*) are representing the lane markings in one frame, but are used as prior distribution in the next frame. In the new frame the lane markings might have moved slightly (because the vehicle moves). The particles need to move the same distance in order to be a valid prior distribution in the new frame.

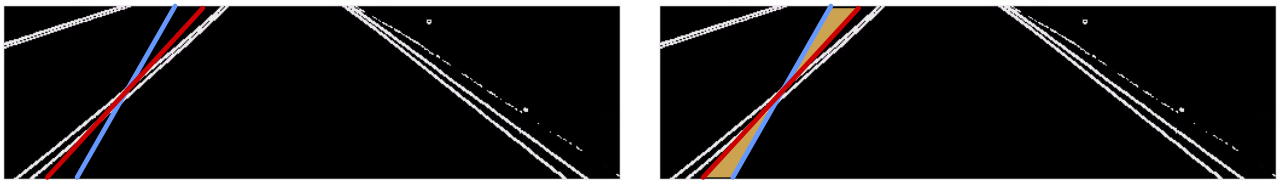
The distance the lane markings shift is not known. Therefore the particles are shifted by a random value sampled from a normal distribution with mean $\mu_{shift} = 0$ and standard deviation $\sigma_{shift} > 0$. $\mu_{shift} = 0$ indicates that we expect no shift in an optimal case and $\sigma_{shift} > 0$ accounts for a deviation from the optimal case.

Importance weight update. For each particle of the prior distribution the importance weight is calculated. This is done by applying formula 2.3. As mentioned there the importance weight of a particle is determined by the numerator: $P(Y|X_i) \cdot P(X_i)$. It assesses the likelihood that the predicted particle X_i produces the observation Y. In this work this is determined by fitting the state of the predicted particle to the Gaussian function

$$w^i_{X_t} = \frac{1}{\sigma\sqrt{2\pi}} e^{-(X_I - \mu_f)^2 / 2\sigma_f^2} \tag{3.7}$$

where $\mu_f = Y$. The standard deviation σ_f represents the measurement noise that accounts for a possible error in the assumption that the position of a lane marking does not change between two frames.

The term $X_i - \mu_f = X_i - Y$ in formula 3.7 represents the distance of two lines, the particle X_i and the observation Y. This distance is calculated by summing up the pointwise distance of line X_i and line Y. It yields the area between the two lines, as shown in figure 3.6.



(a) Best line of a previous frame and a shifted candidate line

(b) The area between the two lines determines the importance weight

Figure 3.6: *Calculation of the importance weight.* In (a) the best line from the previous frame and a candidate line after the prediction update are shown. In (b) the area that determines the importance weight of the particle is highlighted in orange.

The evidence $P(Y)$, or marginal likelihood, is required in order to put the importance weight of different particles in relation to each other. The evidence describes the overall probability of the observation Y and is simply calculated as

$$P(Y) = \sum_{i}^{N} P(Y|X_i) * P(x_i) = \sum_{i}^{N} w_i$$

In accordance with formula 2.3 from the section on particle filter, this results in following formula for the importance weight update:

$$w_i^{\text{updated}} = P(X_i|Y) = \frac{P(Y|X_i) \cdot P(X_i)}{P(Y)} = \frac{w_i}{\sum_i^N w_i} \tag{3.8}$$

Resampling. The prediction and importance weight update produce a new set of good lines, where each particle has a normalized importance weight. Finally, a resampling step is performed in order to increase the accuracy of the lane detection algorithm and to prevent a degeneration of the set (compare section 2.1).

The resampling algorithm selects particles from the updated set according to their normalized importance weight and shifts them to a new set with the same number of particles. Since particles are selected according to their importance weight, good particles are more likely to be selected than less accurate particles. A particle can also be chosen multiple times. The resampling algorithm used in this paper is the same as in (Madduri, 2014, p. 57).

3.4.3 Parallelization of the lane tracking

The prediction and importance weight update of one particle is not dependent on other particles. Therefore these parts can be performed in parallel.

An OpenCL kernel was created that carries out the updates on the hardware accelerator. Algorithm 3 shows the kernel in pseudo code. During a normal lane tracking application hundreds of particles are used for each lane marking. Hence, using a hardware accelerator delivers a significant boost in computation speed.

The resampling, in contrast, is dependent on information from all particles and is implemented on the host.

Algorithm 3 Lane Tracking

1: x_top = x_top + sample_from_Gaussian (Prediction Update)
2: x_bottom = x_bottom + sample_from_Gaussian (Prediction Update)
3: s_x = (x_bottom − x_top) · one_divided_by_row_numbers
4: **for all** ROWS R **do**
5: x_current = x_top + R · s_x
6: distance += abs(x_current − x_best) (Importance Weight Update)
7: **end for**
8: w = fit_to_gaussian(distance) (Importance Weight Update)

3.5 Redetection Criteria

Finally a redetection criteria was introduced that ensures the sanity of the lane detection and tracking results. The criteria checks whether the detected lane marking positions are reasonable and in line with the physical properties of lane markings outlined in section 3.3.1. If the detected positions do not match the criteria, an additional detection step is triggered to discover the positions of the lane markings once again.

The criteria checks three different properties of the positions; it ensures that

- lane markings do not cross
- all detected lane markings have a minimum distance, which was chosen in this work to be 20% of the width of the ROI. However the minimum distance can be adjusted by the user and reduced if many lane markings have to be detected or the ROI is very small
- at least 30 % of a lane marking are in the ROI. Again this is an adjustable parameter.

The last check is important for the lane tracking. The algorithm tracks lane markings based on existing estimates and it will do so even if the lane markings moves to the boundary or even out of the ROI. No new lane markings are detected in the lane tracking stage. In many scenarios, for example when a car changes the lane on the highway, one lane marking moves out of the ROI and another one moves in. The third check of the redetection criteria discovers these cases and triggers a lane detection step to discover the new lane marking.

As the lane tracking is a computationally cheaper stage it is desirable to perform it as often as possible and to use the lane detection only when necessary. The redetection criteria defines the situations where a lane detection step is required. With its introduction the lane detection and tracking method of this thesis is complete.

4 Details on the Implementation

The road lane detection and tracking method presented in the previous chapter was implemented in C++. The parts of the method that can be computed in parallel were implemented as OpenCL kernels and are performed on the hardware accelerator (FPGA/GPU).

4.1 Random number generation

The lane detection and tracking algorithm requires normally distributed random numbers to perform its task. In this thesis normally distributed random numbers are created in two steps. First, uniformly distributed random numbers are generated. Second, they are transformed to normally distributed random numbers, using a fast implementation of the ratio of uniform deviates method (Leva and Mitre Corporation, 1992).

The random number generator MWC64X, developed by David B. Thomas, is used to generate uniformly distributed random numbers in the range $[0, 1]$. The MWC64X is a small generator that was developed for the use with OpenCL on GPUs. It is very fast, as it requires only six instructions to generate a random number and it also displays a high empirical quality (Thomas, 2011).

It creates a stream of random numbers with a period of 2^{63} and it allows the splitting of this stream into independent, non-overlapping substreams with smaller periods. This is desirable for the algorithm of this work, as it uses up to several thousand OpenCL work items per iteration and hence several thousand streams may be required.

In the current work one stream of random numbers is initialized and split into substreams with a period of 2^{40}. This allows the processing of far more than 24 hours of video, even in a worst case scenario where 1000000 random numbers per frame are used.

Numerous changes were made to both the MWC64X and the ratio of uniform deviates method, in order to optimize them for the use on FPGAs. Operations that are costly on FPGAs, such as modulo or floating point operations, were removed and replaced wherever possible.

4.2 Lane detection on multiple hardware accelerators

The lane detection and tracking method presented in chapter 3 can be used in driver assistance systems in future vehicles that incorporate hardware accelerators. As such, it will have to fulfil special safety requirements. Driver assistance systems have an increasing role in assuring driver and road safety and they have to be protected against sudden, unexpected failures. On measure to assure the operability of a DAS using a hardware accelerator is the integration of a back-up device that takes over in case the main device fails. Therefore, an automotive algorithm that uses hardware accelerators has to be able to incorporate multiple, possibly heterogeneous hardware accelerators.

The current work was implemented with that idea in mind and is able to integrate multiple hardware accelerators simultaneously. In addition, the use of more than one hardware accelerator may lead to an increase in computation speed and robustness, as more particles can be used for the lane tracking. The actual impact of several accelerators is investigated in this work and will be presented in section 5.5.1.

The communication between host and devices is organized in a server-client model. The host distributes the workload between the devices and collects the results when they are ready. A more dynamic, direct communication between the devices is not implemented, as OpenCL does not yet support this feature. Figure 4.1 illustrates the processing flow when multiple hardware accelerators are used. The distribution of the workload is kept adjustable. The user defines how many particles are processed on each device and the distribution may be changed at runtime.

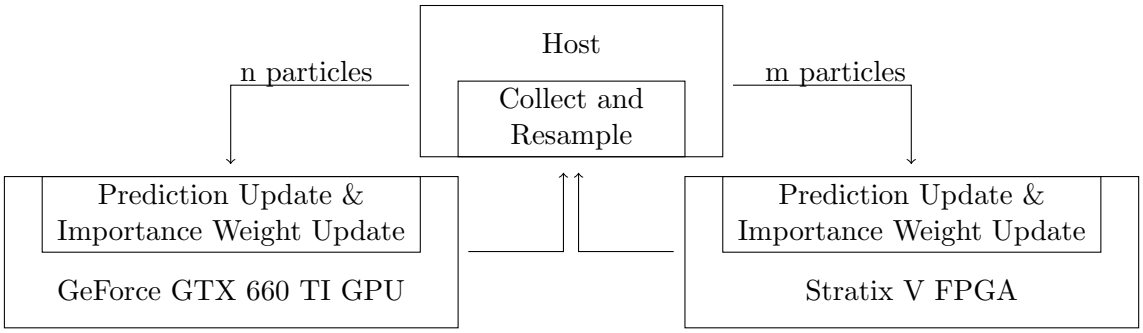


Figure 4.1: *Processing flow of the particle filter when multiple hardware accelerators are used.* The user defines how many particles are processed on each accelerator.

4.2.1 Installable client driver loader (ICD loader)

The main challenge in implementing support for multiple hardware accelerators was the missing installable client driver loader (ICD loader) in Altera's OpenCL (AOCL) implementation version 13.0sp1. As a consequence an application using AOCL 13.0sp1 may integrate multiple FPGAs manufactured by Altera, but not devices from other vendors.

In order to use the NVIDA GeForce GTX 660 TI GPU and an Altera Stratix V FPGA together, a basic ICD loader was developed and implemented. It employs the C programming language library *dl* that is used on Linux to load libraries at runtime.

The self-made ICD loader allows to load Nvidia's and Altera's OpenCL implementations at the same time. All essential methods were encapsulated in handles to avoid clashes, when methods from different libraries but with the same name are called. The use of this self-made and high level ICD can be expected to slow the algorithm down, but its impact proved to be negligible.

5 Results

The previously developed algorithm was subjected to comprehensive accuracy and efficiency testing. The average processing speed was determined for the overall algorithm and the individual parts of the algorithm, as was the memory consumption. Finally, the performance of this algorithm was compared to a preceding work presented in (Madduri, 2014).

5.1 Accuracy of lane detection and tracking

The algorithm was tested with varying testing conditions: It was evaluated on twenty recorded videos, on independent benchmark datasets retrieved from (Fritsch, Kuehnl, and Geiger, 2013) and it was used in multiple live test drives.

The videos were recorded by the author and are used to show the functionality and robustness of the algorithm throughout a wide range of different road scenarios. This includes drives at day and night, with light to heavy traffic, in cities, on highways as well as during varying weather conditions. The videos were recorded with a Samsung S3 mobile phone camera at a resolution of 640x480 pixels. This is a low resolution, which makes it harder to detect and track the lane markings. Nevertheless, the algorithm detected and tracked the lane markings accurately and reliably.

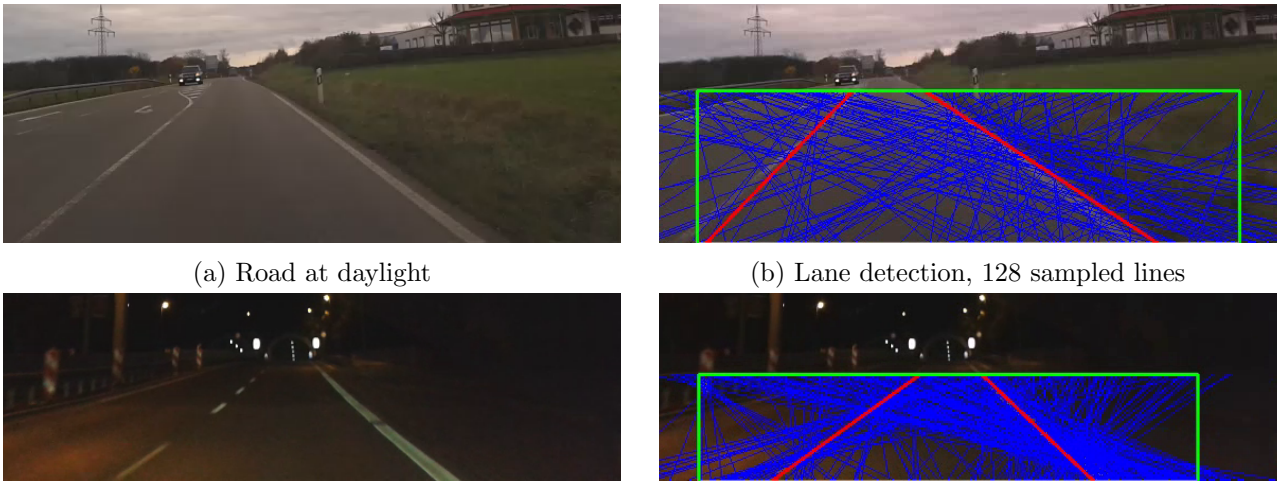
5.1.1 Pre-processing

The pre-processing has only one adjustable parameter, the threshold-value. All other pre-processing parts are fixed operations on pixels. The tests on the videos were performed with a threshold-values in the range of $[50, 75, 100, 125, 150]$ and a threshold value of 50 delivered very good results, both at day and at night. Hence the following tests are performed with a threshold-value of 50, if not indicated otherwise.

5.1.2 Lane Detection

Output examples of the lane detection are displayed in figure 5.1. In figure 5.1a a road is shown at daylight and figure 5.1b shows the same scene after the algorithm detected the lane markings, using 128 randomly sampled lines per marking. It can be seen that the lines (in blue) cover the entire ROI (in green), leaving only small gaps between them. Further, the lines have various orientations, ensuring that also oddly oriented lane markings are detected. The algorithm selected the two best candidate lines (in red) to represent the lane markingsand they are a perfect match. Figures 5.1c and 5.1d show a city road at night and the corresponding output of the lane detection. Again the lane markings are detected, this time using 256 sampled lines per marking.

Two observations can be made in figure 5.1. First, the algorithm detects the lane markings accurately, irrespective of whether the videos were recorded at day or night. That held true throughout all recorded videos and test drives. Second, the algorithm delivers an accurate result, no matter if 128, 256 or more sampled lines are used. If the number of sampled lines is reduced below 128 lines, however, the lane detection algorithm might fail in some cases. The minimum number of lines required



(a) Road at daylight

(b) Lane detection, 128 sampled lines

(c) City road at night

(d) Lane detection, 256 sampled lines

Figure 5.1: *Output of the lane detection.* The lane markings are successfully detected at day and night

to detect the lane markings is dependent on the width of the ROI and the testing revealed that a total of $N_{lines} = \text{WIDTH}_{ROI}/2$ lines per lane marking provides a reliable and accurate detection.

5.1.3 Lane tracking/particle filter

The lane tracking is performed in most frames and therefore its accuracy is of crucial importance. The results of the particle filter are mainly influenced by two parameters: The number of particles that is used to track a lane marking and the standard deviation σ_f, which determines the weight of the particles and defines how far the candidate lines/particles are spread around the *best line*. After extensive testing the latter was chosen to 15 % of the ROI width, as it ensured tracking in a wide range of scenarios. In a future work the results may be improved by adapting σ_f to specific situations.

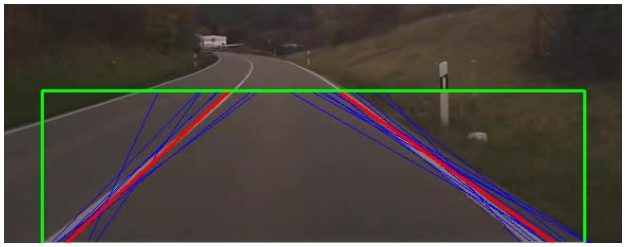
The number of particles is correlated to the computational effort. This makes it desirably to find the minimum number of particles that assure accurate and robust lane tracking. Figure 5.2 shows the output of the particle filter, using varying numbers of particles. Figure 5.2a depicts the original scene and images 5.2b to 5.2f show the output using 16, 32, 64, 128 and 256 particles, respectively. A first observation is that the particle filter tracks the lane markings very accurately, even with only 16 particles. A closer look at figure 5.2b reveals, however, that the tracking with 16 particles is not robust. Only few particles match the lane markings and hence the particle filter is likely to lose track, if the environment changes abruptly.

Increasing the number of particles to 32 (figure 5.2c) improves the particles' average accuracy (and hence the robustness) and with 64 particles (figure 5.2d) the particles form an optimal beam around the lane markings, ensuring their tracking even in a worst-case environment. The use of 128 and 256 particles adds to the algorithm's robustness, but the additional computational effort is unnecessary, as 64 particles are sufficient for tracking.

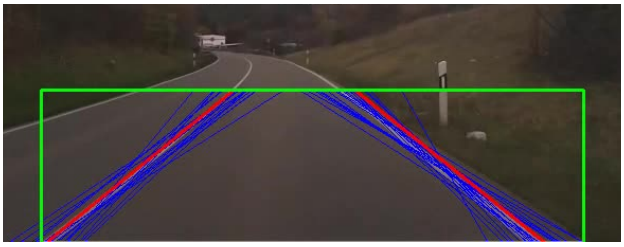
5.2 Performance in challenging environments

When driving in cities, on highways or other roads many exceptional situations might occur, where lane markings are interrupted, bended, blurred, only partly or not at all visible. The following two section present the algorithm's output in these challenging environments.

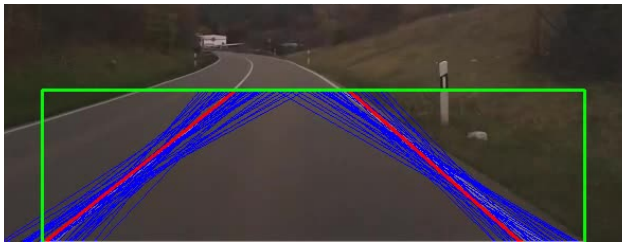
(a) Original image



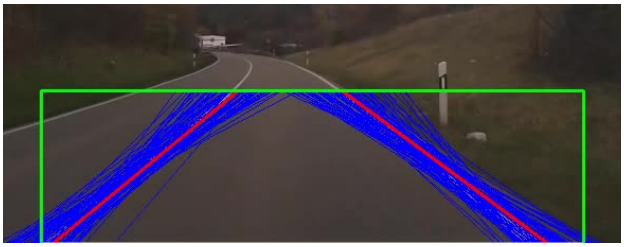
(b) Tracking with 16 particles



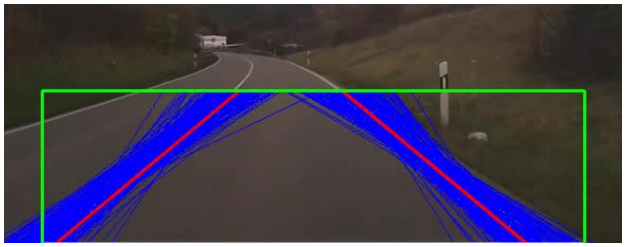
(c) Tracking with 32 particles



(d) Tracking with 64 particles



(e) Tracking with 128 particles



(f) Tracking with 256 particles

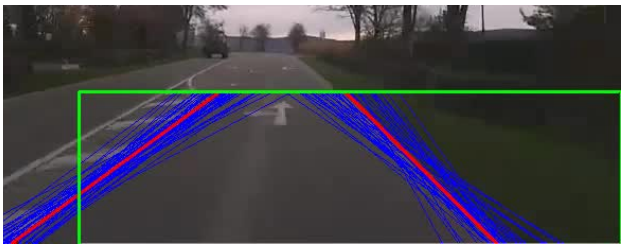
Figure 5.2: *Output of the particle filter for a varying number of particles.* 64 particles are required to deliver an accurate and robust result as shown in (d).

Figure 5.3 displays a selection of images, where the lane tracking is not trivial and the algorithm's results. In 5.3a misleading lane markings are present. The algorithm proved very robust against these kind of disturbances (as it employs estimates from previous images) and tracks the lane markings accurately. Figure 5.3b shows the results of the algorithm for an image displaying a turn. Even though the markings in this image are far from being parallel, they are tracked successfully.

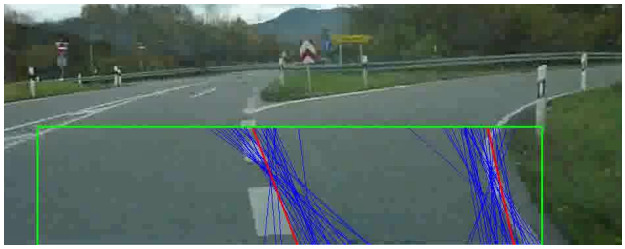
The accuracy of the algorithm does not depend on the number of tracked lane markings, as figure 5.3c illustrates. Neither do negative implications occur, if detection and tracking take place at day or night, which is emphasized in figures 5.3d and 5.3e. A specific challenge at night is the presence of disturbing light beams (5.3e), but the particle filter delivers even then reliable results.

Many more scenarios can be imagined and a more profound testing would be required to further consolidate the algorithms robustness. The tests performed in this thesis, however, indicate a very accurate detection and tracking throughout a wide range of environments.

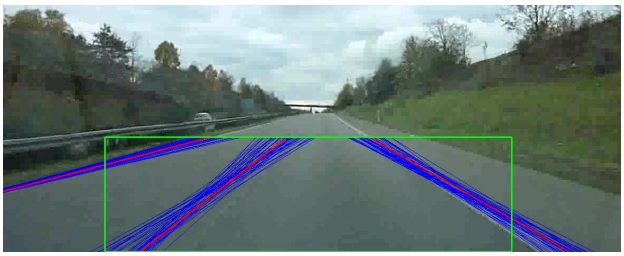
Figures 5.3f -5.3h show the application of the redetection criteria, when the algorithm looses track of a lane marking. The figures depict a car changing a lane on the highway. In figure 5.3f the algorithm looses track of the right lane marking, as it moves out of the ROI. The lane detection is immediately triggered and detects the new lane markings (figure 5.3g). Then the algorithm continues to track the newly detected lane markings (figure 5.3h).



(a) Tracking if misleading marking is present

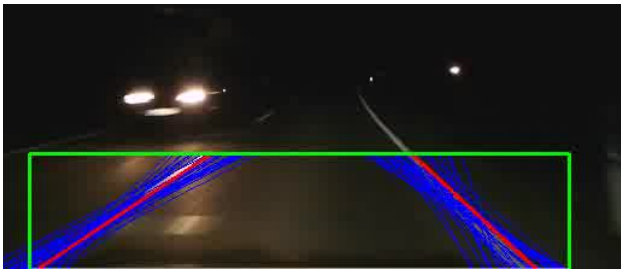


(b) Tracking at turns

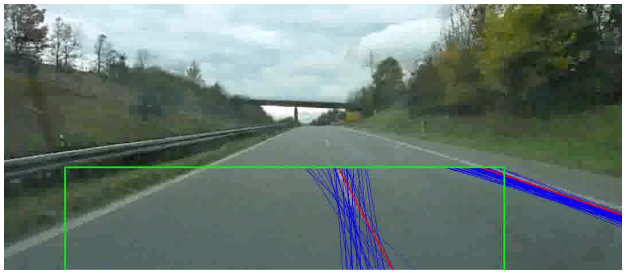


(c) Tracking of multiple markings

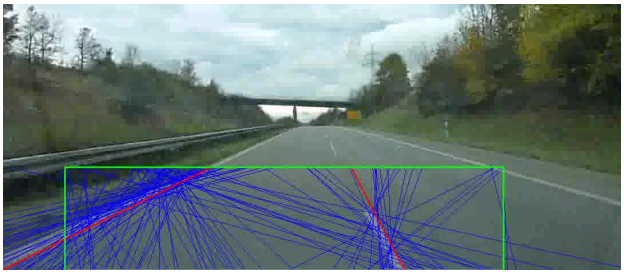
(d) Tracking at night



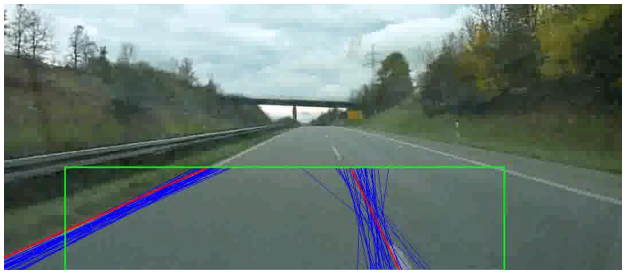
(e) Tracking when a disturbing light beam from another car is present



(f) Redetection 1: Loss of track of the right marking when changing lanes



(g) Redetection 2: Successful detection of new markings



(h) Redetection 3: Continued tracking of the lane markings

Figure 5.3: *Output for road traffic situations that present a challenge to lane tracking algorithms.* The particular situation is described below each figure. The current work was capable of detecting/tracking lane markings in all these situations. Figures (f)-(h) show a car changing a lane on the highway. Consequently the algorithm loses track of the right lane marking (f), performs a successful redetection (g) and continues to track the new lane markings (h)

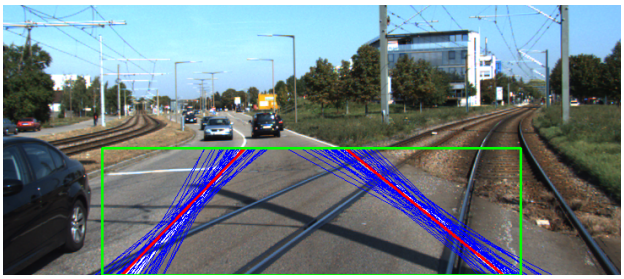
5.3 KITTI-ROAD Dataset

The algorithm's performance was assessed with an independent dataset, the KITTI-ROAD benchmark dataset for lane detection algorithms (Fritsch et al., 2013).

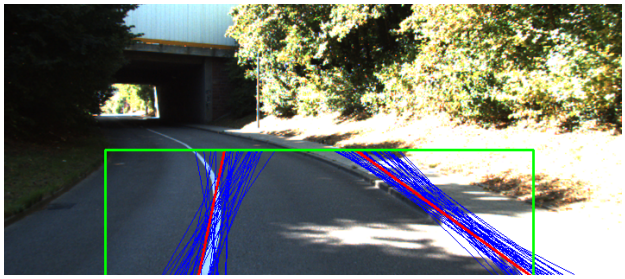
The KITTI-ROAD dataset consists of three independent subsets, the *urban marked two-way road* subset (96 frames) and *urban marked multi-way road* subset (94 frames) and the *unmarked road subset* (100 frames), of which the first two were used. According to the authors the KITTI-ROAD dataset was recorded on five different days in a urban environment with low traffic density and displays a high variability (Fritsch et al., 2013, pp. 2-3).

The current work was able to detect and track the lane markings in 92 % of the images. Figure 5.4 displays the output for some especially challenging images. The situation is described below each figure.

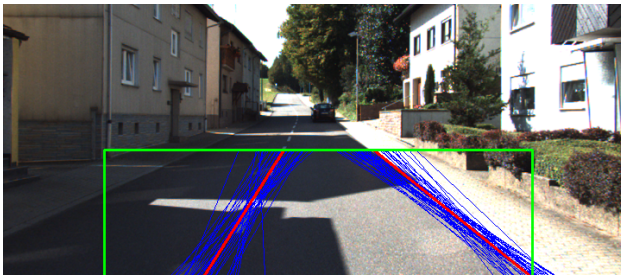
The algorithm only failed to detect/track the lane markings, if they created no edges in the Sobel filter. On the other hand, even the pavement was often enough to create edges and allow the detection/tracking (compare figures 5.4e, 5.4f and 5.4g).



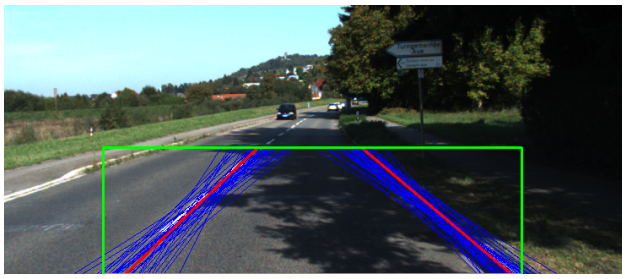
(a) Presence of misleading edges from tracks



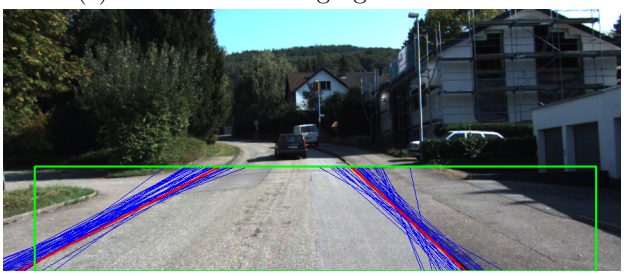
(b) Tracking in sharp bend



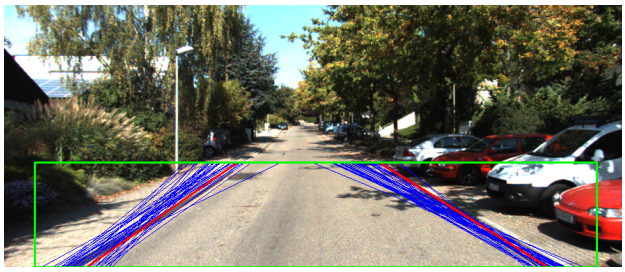
(c) Presence of strong light and shadows



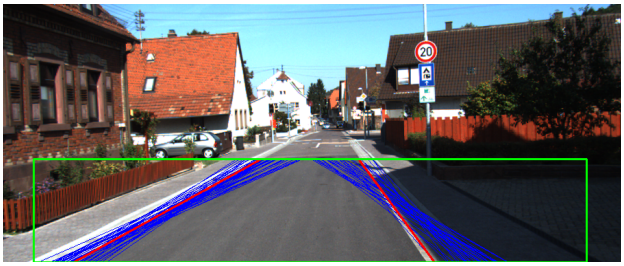
(d) Misleading edges by shadows



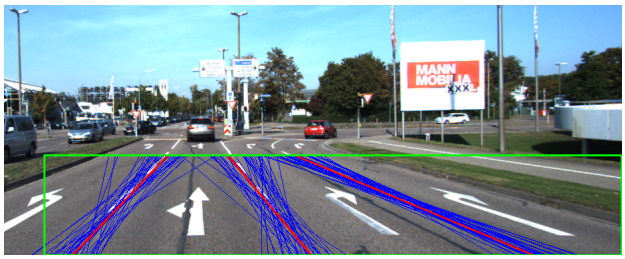
(e) Tracking, when there are no markings



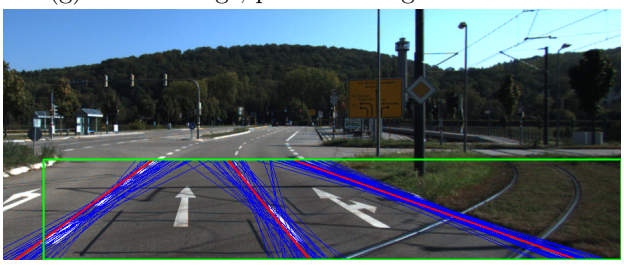
(f) No markings and shadows



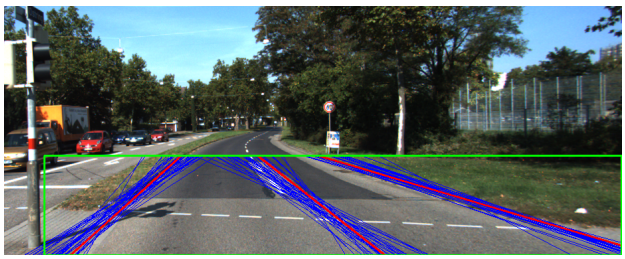
(g) No markings, pavement edges are tracked



(h) Tracking of three lanes in city



(i) Three lanes, disturbing edges from tracks



(j) Three lanes, no markings on the right

Figure 5.4: *Output for the KITTI-ROAD dataset.* All images are taken from (Fritsch, Kuehnl, and Geiger, 2013). The images show a collection of especially challenging scenarios for lane detection/tracking during daylight and the very accurate results of the current work. Each scenario is outlined below the respective figure.

5.4 Test drives

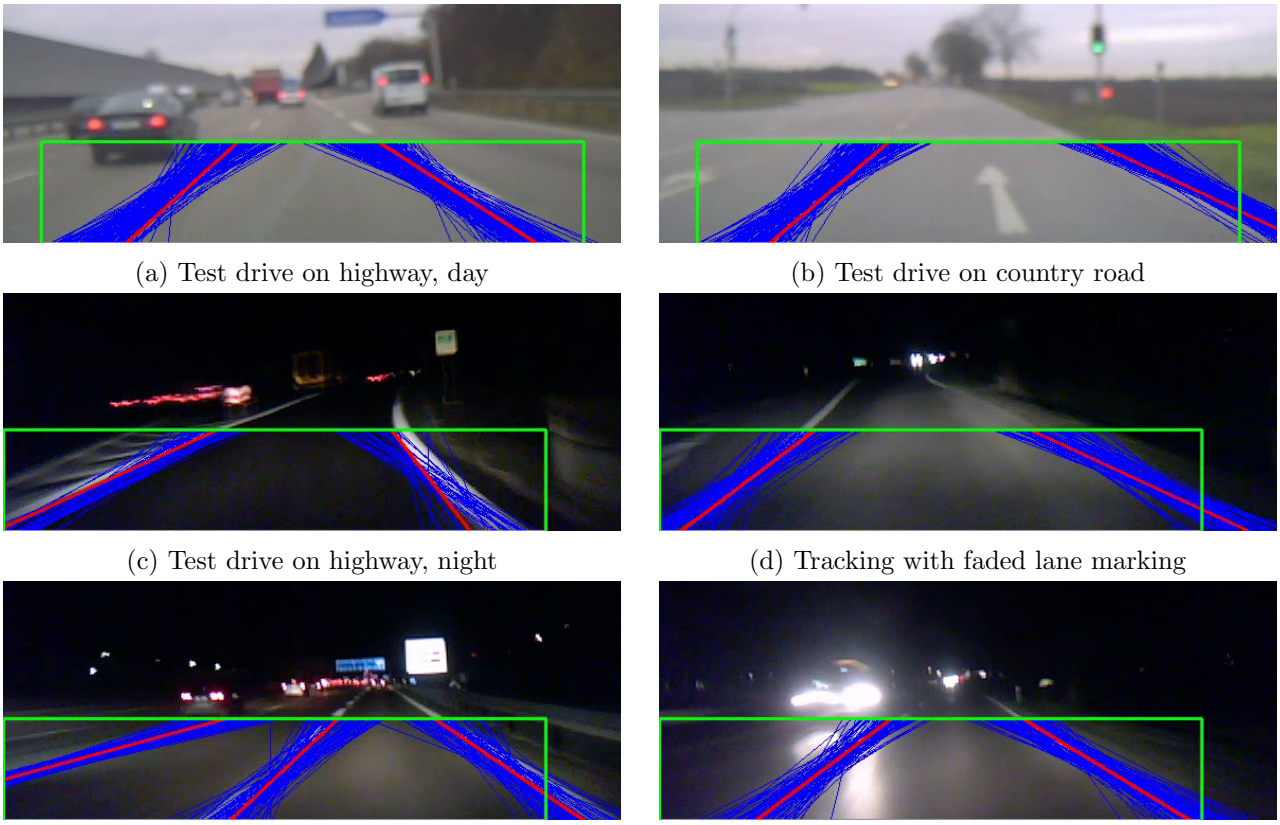
Finally, a number of test drives were carried out to examine the algorithm's performance under realistic conditions. A webcam with a resolution of 640x480 pixels was connected via Ethernet to the Altera Cyclone V SOC. It provided a live stream that was processed by the Cyclone V and the results were displayed in real time on a screen.

The algorithm detected and tracked the lane markings very robust throughout the tests. Strong vibrations of the car (and the person holding the camera) had no negative impact on the results and neither had blurred images (the road was recorded through the front window). Figure 5.5 presents a number of snapshots from the test drives. They, again, show that the algorithm delivers accurate results no matter whether driving at day (5.5b), night (5.5c), on highways (5.5a) or on country roads (5.5d). The performance was also excellent if lane markings were only partly visible (5.5d), multiple lanes were tracked (5.5e) or disturbing light beams were present (5.5f).

5.5 Processing speed

The processing speed is the most important characteristic of a lane detection/tracking algorithm, besides the accuracy. The current work is aimed at the use in a future DAS, a time critical application, and should therefore deliver its results as fast as possible.

The average processing speed is dependent on two factors, the size of the ROI and the number of particles. The ROI size determines the time that the preprocessing and particle filter require and the number of particles solely influences the speed of the particle filter. Figure 5.6 depicts the algorithm's average processing speed at an ROI size of 72x512 pixels for a varying number of particles. The Nvidia GeForce GTX 660 TI GPU delivers with 650 to 578 frames per second (fps) the fastest computation.



(a) Test drive on highway, day

(b) Test drive on country road

(c) Test drive on highway, night

(d) Tracking with faded lane marking

(e) Tracking of three lane markings

(f) Tracking despite disturbing light beam

Figure 5.5: *Snapshots from test drives.* (a) and (b) were recorded during tests drives at day, (c) - (f) at night. In (d) the right lane marking is barely visible, in (e) three lane markings are tracked and in (f) a disturbing light beam is present. The results remain accurate.

This was expected as it has the highest computational power of the tested devices. It is followed closely by the Altera Stratix V FPGA with 540 to 353 fps and after a big gap by the Cyclone V FPGA with 38 to 33 fps.

Influence of the number of particles. Figure 5.6 demonstrates that the number of particles has only a minor influence on the processing speed. On all three devices the average frames per second decrease by less than 10 % if the number of particles is risen from 8 to 128. If the number of particles is further increased to 256 or 512 the particles' impact on the execution time grows stronger. But as 64 particles proved sufficient to deliver accurate estimates, there is no need to use more particles.

Influence of the ROI size. The size of the ROI has a much stronger impact on the computation time. In figure 5.7 the processing speed is shown for three different ROIs (72x512 pixels, 96x512 pixels and 144x1024 pixels). Increasing the ROI from 72x512 pixels to 96x512 pixels reduces the processed frames per second by 40-50%, depending on the device. A further increase to 144x1024 pixels yields an additional performance reduction of 30-40%. This is a sharp decline in the computation speed, which gives rise to the maxim of *choosing the ROI as small as possible.* This is also a possible approach for future works to further improve the performance.

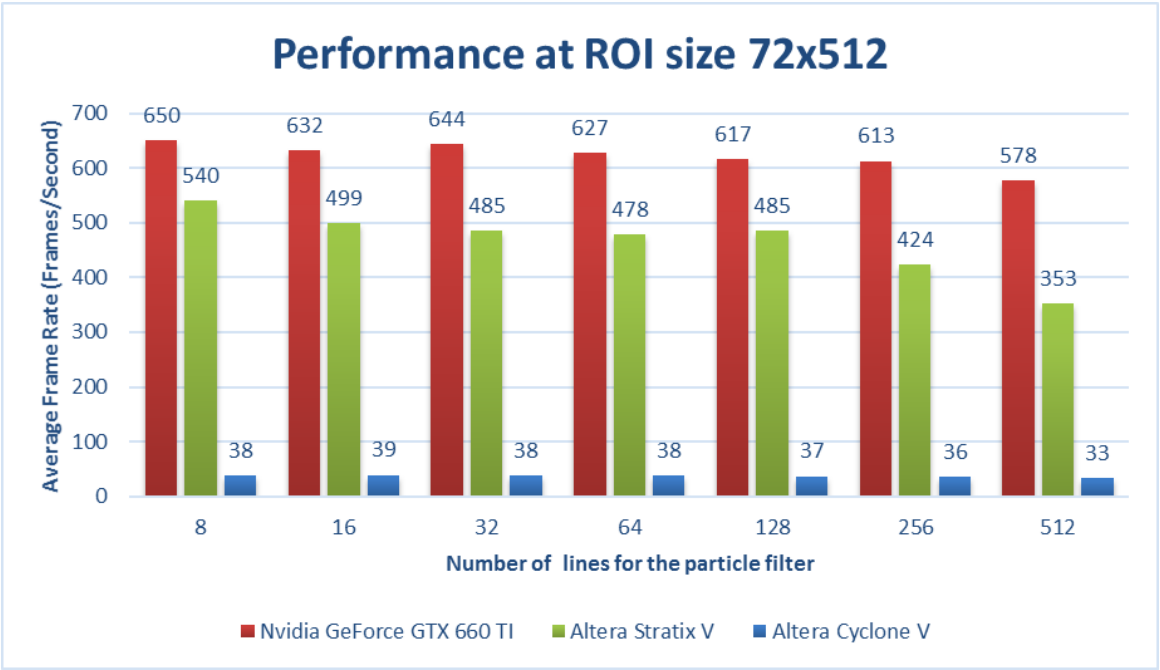


Figure 5.6: *Performance at a ROI size of 72x512 pixels.* The performance is shown using the Nvidia GeForce GTX 660 TI GPU (red), the Altera Stratix V FPGA (green) and the Altera Cyclone V SOC (blue).

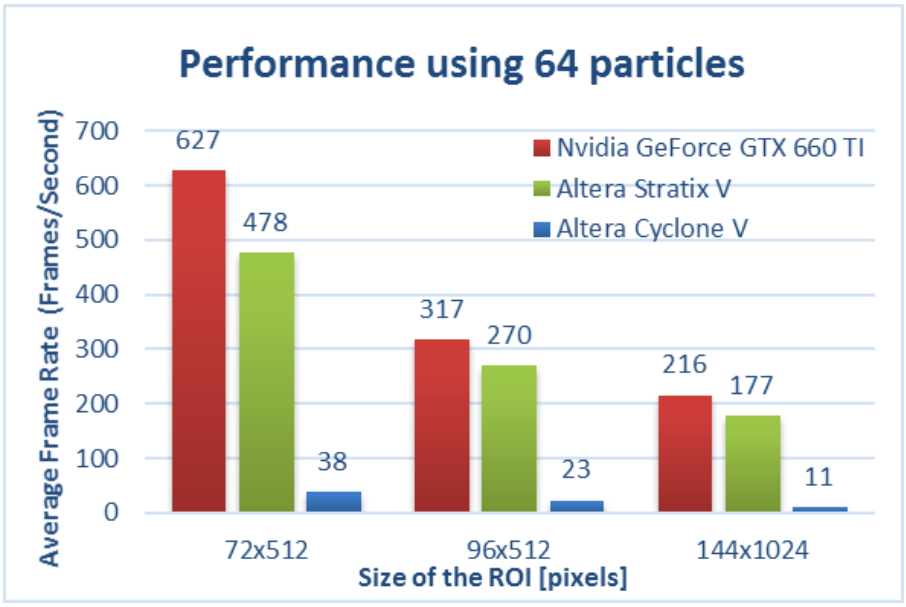
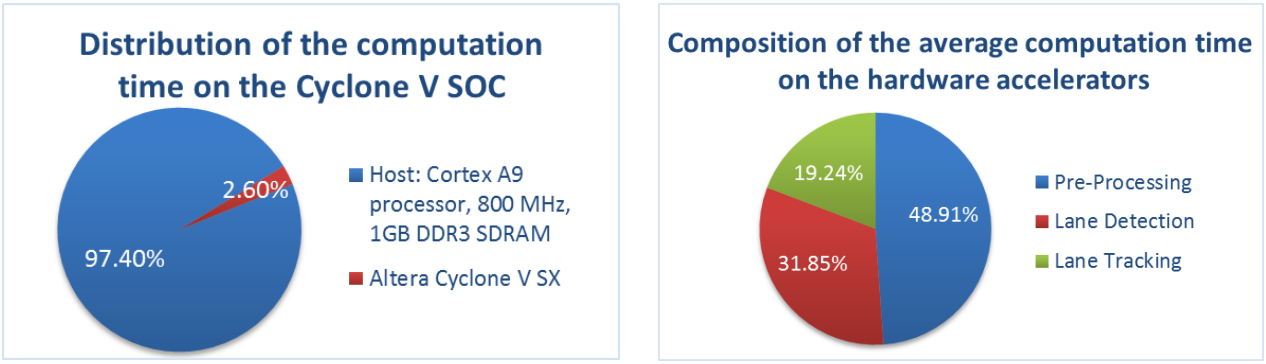


Figure 5.7: *Performance using 64 particles and varying ROI sizes.* Three ROI sizes are compared: 72x512 pixels, 96x512 pixels and 144x1024 pixels. A sharp performance decrease is detected with an increasing ROI size.



(a) Distribution of the computation time between host and device on the Altera Cyclone V SOC

(b) Distribution of the computation time between pre-processing, lane detection and lane tracking

Figure 5.8: *Composition of the computation time*

Composition of the computation time. In a setup using 64 particles and a ROI of 72x512 pixels the algorithm requires 1.59 ms per frame on the GPU, 2.09 ms per frame on the Stratix V FPGA and 26.31 ms on the Cyclone V FPGA (compare figure 5.6). It was expected that the Cyclone V would process the videos slower, but the difference is immense. The reason is explained by figure 5.8a. The chart shows that the calculations on the host system of the Cyclone V SOC require 97.40 % (which are 25,63s) of the overall computation time. A lot of time is consumed by loading images and other peripheral tasks. This indicates that the performance is reduced drastically by the slow host system running on the ARM Cortex-A9 processor and not by a design fault or the Altera Cyclone V FPGA itself.

Figure 5.8b shows the average distribution of the computation time on the hardware accelerators. The distribution on all three devices was similar and therefore an average is presented. The pre-processing requires with 48.91 % the biggest amount of computation time, further emphasising the need to minimize the ROI size. Lane detection was performed in the tests on only 3% of the frames but it still accounts for 31.85 % of the processing time. This justifies the use of the particle filter that required only 19.24% of the total time, but processed 97% of the frames.

5.5.1 Computation on multiple devices

In section 4.2 the use of multiple devices for the lane tracking and detection algorithm was outlined and a possible acceleration of the computation was predicted. The actual impact of multiple devices on the computation speed was tested using Altera's Stratix V FPGA and Nvidia's GeForce GTX 660 TI GPU in combination. The results are summarized in figure 5.9.

The figure shows that combining the two devices delivers no speed-up in computation when 64 particles are used. On the contrary, if one device was used exclusively, the execution was sometimes even faster (4.74s on the GPU, 6.13 on the FPGA) than in combination. With 6.29s the slowest computation occurred, when 80 % of the particles were computed on the FPGA. This stems from waiting delays that occur when the host collects the results from the device.

Increasing the number of particles to 2048 changed the results significantly. Now the computation was accelerated if more particles were computed by the GPU. The incorporation of the GPU helps to deliver results in a computation time that the FPGA alone is not capable of. It can also be interpreted as the FPGA slowing the GPU down, even if only very few particles are computed on it. Using only the GPU yielded the fastest computation, indicating that the GPU is computationally far more powerful

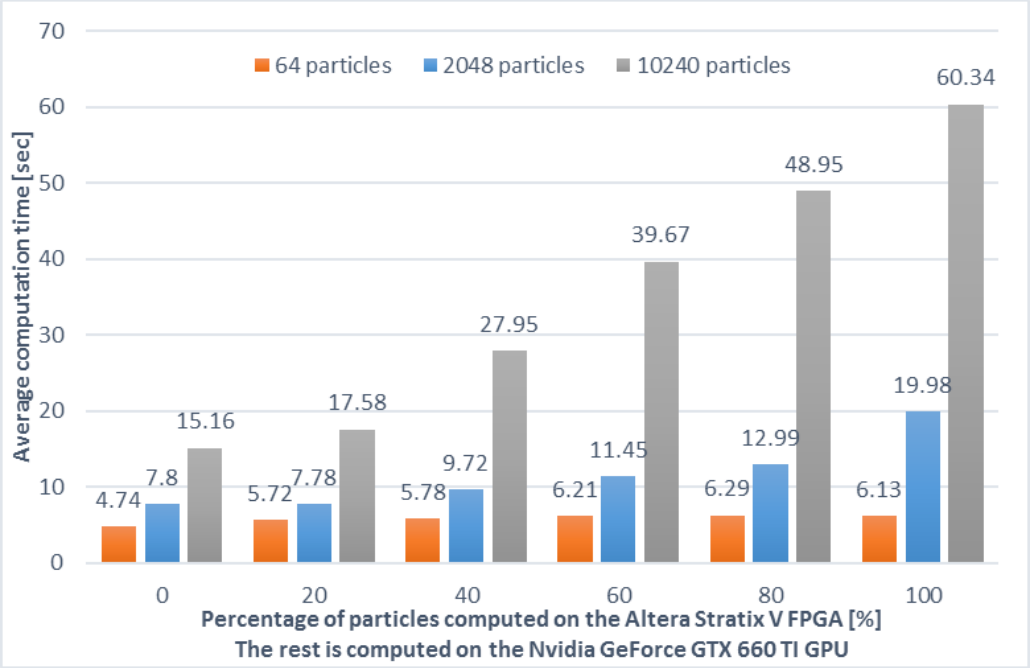


Figure 5.9: *Performance using Altera's Stratix V FPGA and Nvidia's GeForce GTX 660 TI in combination*

than the FPGA. Another increase of the particles to 10240 delivered similar results and consolidate this explanation.

Summarizing the above, the use of multiple devices delivers no speed-up in computation for the current work, where 64 particles are sufficient for tracking. It accelerates the computation if much more particles are used, which might be interesting for other applications.

5.6 Comparison to previous works

The current work was extensively compared to a preceding work presented in (Madduri, 2014). Both algorithms use a particle filter (of different design) to track the lane markings. The similarity of the algorithms allows for a comprehensive comparison using the *Nvidia GeForce GTX 660 TI GPU* and the *Altera Stratix V FPGA* (Madduri's algorithm cannot be ported to the *Altera Cyclone V FPGA*).

The algorithms were tested on two videos, which Madduri recorded for his thesis (compare Madduri, 2014, pp. 71,74). Table 5.1 shows and explains the parameters that were used in the tests. The algorithms could not be compared on additional data, as Madduri's algorithm is tailored for a certain set of videos used in his thesis and hence not portable. Throughout the tests the performance of the algorithms was documented and compared with focus on three crucial aspects:

1. The accuracy of the lane detection/tracking algorithm

2. The resource consumption of the algorithm, including the data transfer between host and devices

3. The processing speed (average frames per second that can be processed by each algorithms)

Property	Value	Explanation
ROI size	108x448 pixels	The size of the region of interest (ROI)
Best lines	2	Number of lane markings that are tracked
Particles	16, 32, 64, 128, 300, 512	Number of candidate lines/particles that were used to track each lane marking

Table 5.1: Parameters that were used for comparing the performance of the current work to Nihil Madduri's algorithm

5.6.1 Accuracy

The output of both particle filters using a varying number of particles is presented and compared in figure 5.10. Madduri's algorithm required in the tests a minimum of 128 particles to deliver accurate estimates of the lane markings. This is shown by figures 5.10a, 5.10c and 5.10e where his particle filter fails to track at least one marking.

Further, the previous work often loses track of lane markings, even if 128 or more particles are used. This is for example the case when the orientation of lane markings changes, as shown in figure 5.10g. Other changes is the scenery, such as bright street signs or disturbing light beams frequently led to a failure as well.

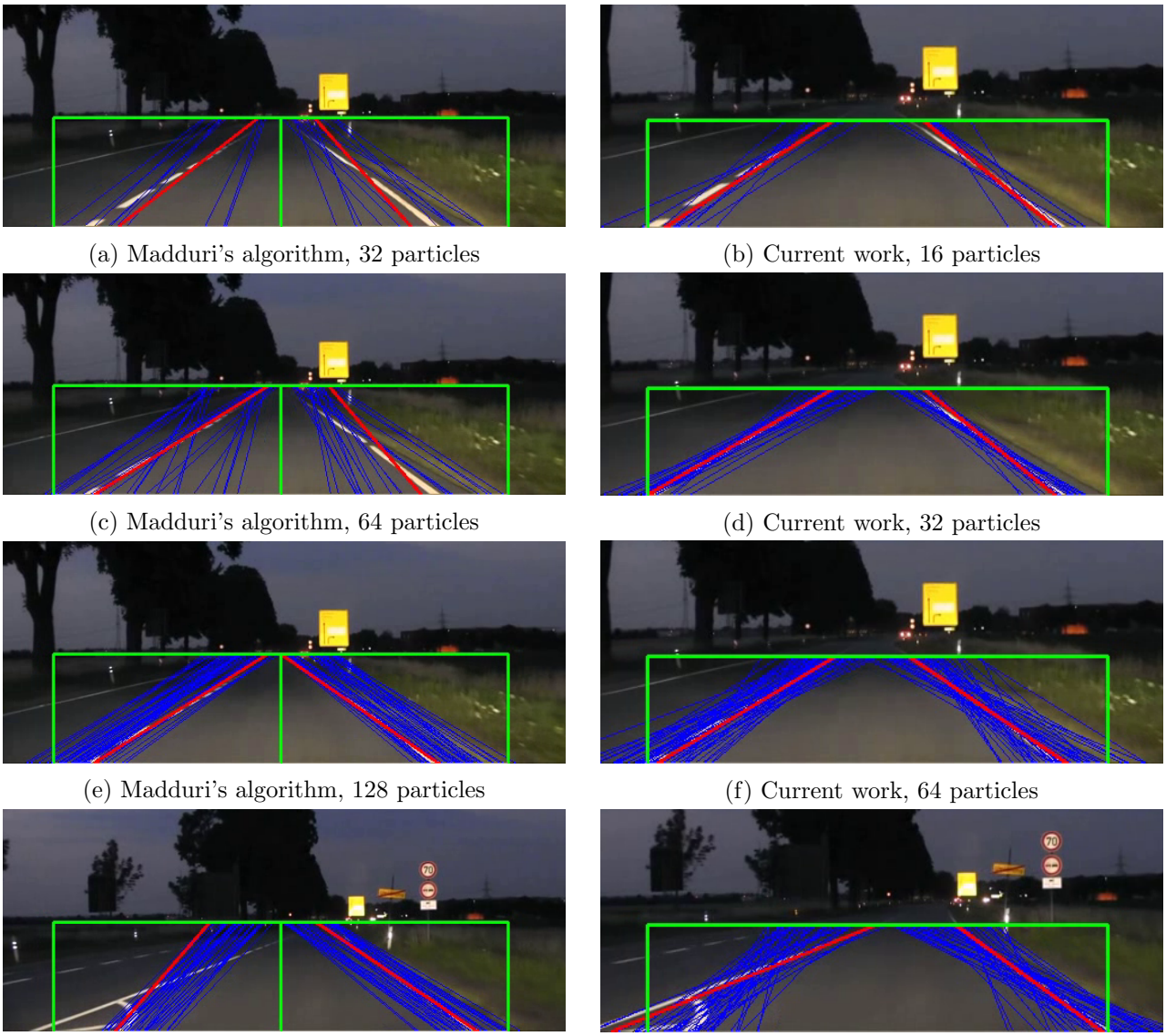
This failures are caused by a flaw in the design of Madduri's particle filter. It only uses the angle of the particles (candidate lines) as resampling criteria. The resulting particles are parallel lines, but their distance to the actual lane markings is not limited and might be big.

Since the particles are parallel, it is impossible to accurately track a lane marking with a orientation that is not parallel to the estimates of the previous frame. Moreover, distant light beams or other areas with high-intensity-pixels are frequently confused as lane markings, as particles are not limited in their distance to the previous best line. This can be seen in figure 5.10c, where the particles are spread throughout the ROI.

In addition, the tests showed that once Madduri's algorithm lost track, it took it some time (at least one or two seconds) to find back to the actual lane marking, if it was able to at all. This is unacceptable for a DAS system, which might need to react very fast.

The current work, on the other hand, delivered very accurate results, already with 32 particles. Figures 5.10d and 5.10f show the algorithm tracking the lane markings and the results are very accurate, irrespective of whether 32 or 64 particles are used. This is tantamount to an enormous increase in accuracy at a much lower computational effort. Much less particles are used to achieve the same or an even better result.

In addition to that, the robustness of the algorithm exceeds that of the previous work by far. As shown in figure 5.10h the current works tracks both lane markings successfully when changes in the orientation of lane markings occur. This also held true for other disturbing influences, such as additional lane markings, disturbing light beams and blurred images. The improved resampling criteria, presented in section 3.4.2, ensured a robust lane tracking throughout all these scenarios.



(a) Madduri's algorithm, 32 particles

(b) Current work, 16 particles

(c) Madduri's algorithm, 64 particles

(d) Current work, 32 particles

(e) Madduri's algorithm, 128 particles

(f) Current work, 64 particles

(g) Madduri's algorithm, tracking fails

(h) Current work, tracking successful

Figure 5.10: *Accuracy of the current work in comparison to Madduri, 2014* Figure (a), (c), (e) show the accuracy of Madduri's work at 32, 64 and 128 particles, respectively. At least 128 particles are necessary to ensure reliable tracking. Figure (b), (d), (f) show the accuracy of the current work. 32 particles are already sufficient for the lane tracking. Figure (g) shows a case where Madduri's algorithm, fails to track a lane marking, as only parallel lines are resampled. Figure (h) shows the same scene and the current work successfully tracks both lane markings.

5.6.2 Memory consumption

The memory consumption is an important characteristic, which influences processing speed and applicability of the algorithm. A lane detection algorithm has to be both compact and fast. It should consume as less memory resources as possible to allow other applications to work and it should still process the frames fast enough for the DAS.

The amount of data that has to be transferred for each frame between host system and device has a strong impact on the overall computation speed: A higher data transfer results in a lower computation speed. The current work could reduce the resource consumption on the host system and the device as well as the data transfer between the two systems decisively. Table 5.2 compares the transferred data for the test conditions of table 5.1. It can be observed that the current algorithm has to transfer 71.94% less data than the previous work. This obviously translates into an increase in processing speed, which will be presented in the next section.

The enormous reduction of transferred data between host and device is achieved by a novel representation of candidate lines. Madduri chose to represent lines by an x-intercept at the top of the ROI and an angle. In addition he stored every x- and y-value of the candidate line. This accumulates to an considerable amount of (unnecessary) data that has to be stored. As described in section 3.2 this work took a more compact approach, resulting in only 3 bytes memory storage per lane, while keeping the same amount of information. This can be seen in table 5.2: The main difference of the algorithms' resource consumption is the amount of memory that the candidate lines require.

Kernel argument	**Memory size (bytes)**	
	Madduri's algorithm	**Current Work**
Image	193536	193536
Candidate lines	523200	7200
Best lines	1744	6
Parameters	44	40
Pf_weight	1200	1200
Total	719724	201982

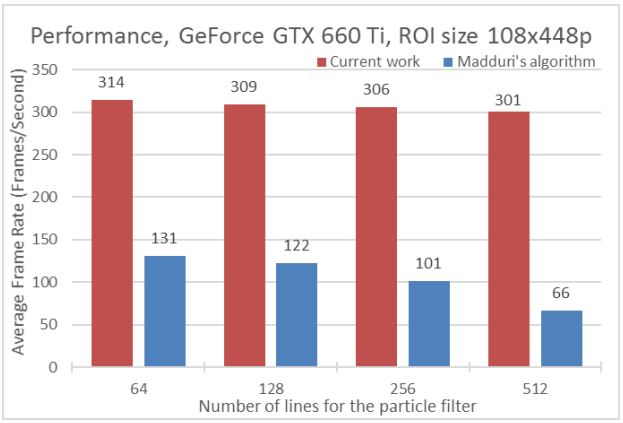
Table 5.2: Comparison of the memory consumption

5.6.3 Processing Speed

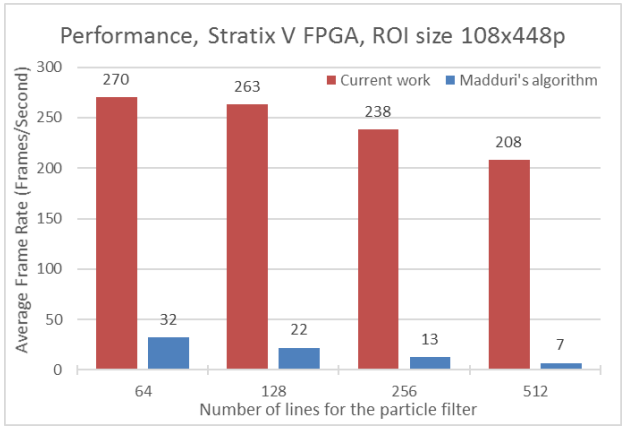
The algorithm of this thesis showed an extremely improved performance. Figure 5.11a displays the processing speed of both algorithms using the NVIDIA GeForce GTX 660 TI GPU with a ROI size of 108x448 pixels. The current work achieved an average frame rate of more than 300 fps, irrespective of the number of particles used. This makes it between 2.4 and 4.6 times faster than the preceding work of Madduri, amounting to an increase of 183 fps if 64 particles per lane marking are used and even 235 fps if 512 particles are used.

When an Altera Stratix V FPGA is used, the performance gap is even wider and the current work executes between 8.4 times (270 vs. 32 fps) and 29.7 times faster (208 fps vs. 7 fps). The enormous acceleration in both cases is due to the following three differences in the algorithm, ordered by their impact on the processing speed:

- The pre-processing is performed in the current work in parallel and in Madduri's work in series.



(a) Performance using the GPU



(b) Performance using the FPGA

Figure 5.11: *Processing speed of the current work in comparison to Madduri's algorithm* The ROI size is in both figures 108x448 pixels. Figure (a) compares the processing speed of both algorithms using a Nvidia GeForce GTX 600 TI GPU. The current work processed videos between 2.4 and 4.6 times faster. Figure (b) compares the processing speed using an Altera Stratix V FPGA. The current work executed between 8.4 and 29.7 times faster.

- The design of the particle filter. In this thesis the particles consume less memory and do not require computationally expensive operations (sinus, cosinus, divisions)
- A more efficient implementation of the random generator, as in the current work the random generator is only initialized once.

The charts in figure 5.11 reveal another difference between the two algorithms. The previous work's performance is strongly dependent on the number of particles. On the NVIDIA GeForce GTX 660 TI the processed frames per second decrease by 50 % (from 131 fps to 66 fps), when the number of particles is increased from 64 to 512. On the Altera Stratix V FPGA the difference is stronger. With 64 particles the algorithm is able to process 32 fps, which is just enough to process a normal video with 30 fps, but with 512 particles the performance reduces by 78% to 7 fps, which is to slow to process a live video.

The current work's processing speed, however, exhibits a much lower dependence on the number of particles. Its performance decreases only by 4 % (314 fps to 301 fps) on the GeForce GTX 660 TI GPU and by 22 %(270 fps to 208 fps) on the Stratix V FPGA. This can be explained by the novel representation of lane markings, which was introduced in this work. It reduced the memory consumption of the particles remarkably, making them insignificant in terms of resource consumption.

6 Conclusion

A vision-based road lane detection and tracking algorithm is developed, implemented using C++ and OpenCL, and tested on two FPGAs and one GPU. A incoming life video stream is processed in two steps, an initial pre-processing step that creates a gradient image of a selected region and removes minor noises, followed by a lane detection or lane tracking step. The lane detection method is a greatly improved version of a novel approach proposed in (Madduri, 2014). It populates the gradient image with randomly sampled lines, weights these lines according to their distance to the lane markings and selects the best fitting lines to represent the lane marking. Lane tracking uses a computationally efficient particle filter to track lane markings. A redetection criteria was introduced that triggers an additional redetection step, if the results are not feasible.

The pre-processing is implemented as an OpenCL kernel and performed on a hardware accelerator. The random sampling and weighting of lines in the lane detection step are moved to a kernel as well as the prediction and importance weight update in lane tracking step.

In various tests with recorded videos, independent datasets and life test drives the algorithm delivered very accurate and robust results using only 64 particles. The tests covered the detection on highways, in cities, with low and high traffic density, at day and night and in many exceptional situations, including at turns, sharp bends, when shadows or disturbing light beams were present and even if lane markings were barely visible. In all cases the algorithm delivered excellent results.

The performance varied with the selection of the ROI size and the hardware accelerator. The Nvidia GeForce GTX 660 TI GPU delivered an extremely fast result with an average of 216-627 processed fps. It was followed by the still impressive performance of the Altera Stratix V FPGA with 177-478 fps and the Altera Cyclone V FPGA with 11-38 fps. In case of the Cyclone V the slow performance can be explained by the significantly slower host system in which the Cyclone V is embedded. The integration of multiple hardware accelerators at the same time delivered no significant acceleration of the computation, mainly because 64 particles are already sufficient for tracking.

The algorithm showed in a comparison to a preceding work an enormous increase in accuracy and robustness, a by 71.94% reduced memory consumption, a 2.4- 4.6 times faster execution on the GeForce GTX 660 TI and a 8.4-29.7 times faster execution on the Stratix V, due to numerous improvements in design and implementation.

6.1 Future Work

A number of measures can be taken to further improve the performance of the algorithm. The remaining bottleneck are the parts of the algorithm that are performed on the host. The selection of the good lines in lane detection and the resampling in lane tracking can be moved to the OpenCL kernels using parallel reduction methods. Then the entire algorithm would run on the hardware accelerator, which would result in a great performance boost as no device-host operations would be necessary. It would also allow to use a hardware description language like VHDL for the implementation on the FPGA, which would deliver a significant boost (but a reduction of flexibility). Other possible improvements are a dynamic selection of the ROI and a situation-specific calculation of the importance weight update, which would further increase the robustness of the algorithm.

Bibliography

Cappe, O., Godsill, S., & Moulines, E. (2007, May). An overview of existing methods and recent advances in sequential monte carlo. *Proceedings of the IEEE*, *95*(5), 899–924. doi:10.1109/JPROC.2007.893250

Cunningham, W. (2011). Bmw adopts nvidia gpu for in-car displays. Retrieved March 9, 2015, from http://www.cnet.com/news/bmw-adopts-nvidia-gpu-for-in-car-displays/

destatis. (2013, July). *Unfallentwicklung auf deutschen strassen 2012*. Brochure. Wiesbaden: Statistisches Bundesamt Deutschland.

Fons, F. & Fons, M. (2012, March). Fpga-based automotive ecu design addresses autosar and iso 26262 standards. *Xcell Journal*, *78*, 20–31.

Fritsch, J., Kuehnl, T., & Geiger, A. (2013). A new performance measure and evaluation benchmark for road detection algorithms. In *International conference on intelligent transportation systems (itsc)*.

Gordon, N., Salmond, D., & Smith, A. (1993, April). Novel approach to nonlinear/non-gaussian bayesian state estimation. *Radar and Signal Processing, IEE Proceedings F*, *140*(2), 107–113.

Leva, J. L. & Mitre Corporation. (1992). A fast normal random number generator. *ACM Trans. Math. Software*, 449–453.

Madduri, N. (2014). *Hardware accelerated particle filter for lane detection and tracking in opencl* (Master's thesis, TU München).

Muyan-Özçelik, P. & Glavtchev, V. (2008). Gpu computing in tomorrow's automobiles. Retrieved March 9, 2015, from http://www.nvidia.com/content/nvision2008/tech_presentations/automotive_track/nvision08-gpu_computing_in_tomorrows_automobiles.pdf

Navarro, C. A., Hitschfeld-Kahler, N., & Mateu, L. (2013). A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, *15*(2), 285–329. doi:10.4208/cicp.110113.010813a

Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., & Phillips, J. (2008, May). Gpu computing. *Proceedings of the IEEE*, *96*(5), 879–899. doi:10.1109/JPROC.2008.917757

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., & Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, *26*(1), 80–113. doi:10.1111/j.1467-8659.2007.01012.x

Peddie, J. (2013). Mobile devices and the gpus inside. Jon Peddie Research.

Thomas, D. B. (2011). The MWC64X Random Number Generator. Retrieved from http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html

Tian, X. & Benkrid, K. (2010, November). High-performance quasi-monte carlo financial simulation: fpga vs. gpp vs. gpu. *ACM Trans. Reconfigurable Technol. Syst. 3*(4), 26:1–26:22. doi:10.1145/1862648.1862656

Yazbeck, F. & Kenny, R. (2012, March). *White paper: reducing power consumption and increasing bandwidth on 28-nm fpgas*. Altera. Retrieved from https://www.altera.com/en_US/pdfs/literature/wp/wp-01148-stxv-power-consumption.pdf