GPU-ACCELERATED ADAPTIVE UNSTRUCTURED ROAD DETECTION USING CLOSE RANGE STEREO VISION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

KADRİ BUĞRA ÖZÜTEMİZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MECHANICAL ENGINEERING

JULY 2013

Approval of the thesis:

GPU-ACCELERATED ADAPTIVE UNSTRUCTURED ROAD DETECTION USING CLOSE RANGE STEREO VISION

submitted by **KADRİ BUĞRA ÖZÜTEMİZ** in partial fulfillment of the requirements for the degree of **Master of Science in Mechanical Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Süha Oral
Head of Department, **Mechanical Engineering** _____

Asst. Prof. Dr. Buğra Koku
Supervisor, **Mechanical Engineering Dept., METU** _____

Assoc. Prof. Dr. İlhan Konukseven
Co-Supervisor, **Mechanical Engineering Dept., METU** _____

Examining Committee Members:

Prof. Dr. Tuna Balkan
Mechanical Engineering Dept., METU _____

Asst. Prof. Dr. Buğra Koku
Mechanical Engineering Dept., METU _____

Assoc. Prof. Dr. İlhan Konukseven
Mechanical Engineering Dept., METU _____

Asst. Prof. Dr. Yiğit Yazıcooğlu
Mechanical Engineering Dept., METU _____

Assoc. Prof. Dr. Afşar Saranlı
Electrics and Electronics Engıneering Dept., METU _____

Date: 16.07.2013

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Kadri Buğra Özütemiz

Signature:

ABSTRACT

GPU-ACCELERATED ADAPTIVE UNSTRUCTURED ROAD DETECTION USING CLOSE RANGE STEREO VISION

Özütemiz, Kadri Buğra
M.Sc., Department of Mechanical Engineering
Supervisor : Asst. Prof. Dr. Buğra Koku
Co-Supervisor : Assoc. Prof. Dr. İlhan Konukseven

July 2013, 125 pages

Detection of road regions is not a trivial problem especially in unstructured and/or off-road domains since traversable regions of these environments do not have common properties unlike urban roads or highways. In this thesis a novel unstructured road detection algorithm that can continuously learn the road region is proposed. The algorithm gathers close-range stereovision data and uses this information to estimate the long-range road region. The experiments show that the algorithm gives satisfactory results even under changing light conditions. In addition to the algorithm structure, the massive parallel implementation on GPU with CUDA is proposed. The speed-up of the CUDA implementation with experiments done is analyzed.

Keywords: Unmanned Ground Vehicle, Point Cloud, Road Detection, Unstructured Roads, Unstructured road detection, Traversability Detection, CUDA, GPU

ÖZ

YAKIN MESAFE STEREO GÖRÜNTÜ KULLANILARAK GPU ILE HIZLANDIRILMIŞ UYUMLU YAPISIZ YOL BULMA

Özütemiz, Kadri Buğra
M.Sc., Department of Mechanical Engineering
Supervisor : Asst. Prof. Dr. Buğra Koku
Co-Supervisor : Assoc. Prof. Dr. İlhan Konukseven

Temmuz 2013, 125 sayfa

Yol bölgelerinin saptanması özellikle yapısız ve/veya arazi bölgelerinde çözülmesi zor olan bir problemdir. Yapısız bölgeler ya da arazi bölgeleri, şehiriçi veya şehirlerarası bölgeler gibi benzer özeliklere sahip olmadığından bu bölgelerde yol bulmak kolay değildir. Bu tez kapsamında yol bölgesini sürekli öğrenebilen yenilikçi bir yapısız yol saptama yöntemi önerilmiştir. Önerilen yöntem yakın çevredeki stereo kamera verisini kullanarak görüntü üzerindeki yol olan bölgeleri saptamaktadır. Yapılan deneyler önerilen yöntemin değişken ışık koşulları altında bile tatmin edici sonuçlar sağladığını göstermektedir. Bu tez çalışmasında önerilen yöntemin yanı sıra GPU üzerinde CUDA dili kullanılarak yapılmış olan paralelleştirme çalışması da sunulmuştur. Paralelleştirme sonucu elde edilen performans yükselmesi deneyler eşliğinde analiz edilip sunulmuştur.

Anahtar Kelimeler: İnsansız Kara Aracı, Nokta Bulutu, Yol Saptama, Yapısız Yol Saptama, Geçilebilirlik Saptaması, CUDA, GPU

To my family

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere appreciation to my supervisors Asst. Prof. Dr. A. Buğra Koku and Assoc. Prof. Dr. E. İlhan Konukseven for their invaluable guidance, advice, and criticism and especially their extreme support that made this study possible.

I would like to express my thanks to my dear friends and colleagues Serkan Tarçın, Akif Hacınecipoğlu, Matin Ghaziani, Emre Akgül and İlker Moral for their friendship and technical support throughout the thesis period. I would also like to thank my friends and collegues Turgay Aydınlılar, Tolga Uzun, Rasim Aşkın Dilan, Burak Şamil Özden, Ali Anıl Şahinci, Mehmet Ali Çiftçi, Kemal Arı, Ömür Deler, Yaser Mohammadi Nazarabad, Hasan Ölmez, Kasım Gönüllü, Gökhan Bayar, Akay Öztürk and Murat Topçu for all the good times we have spent. I want to thank to Muhtar Ural Uluer for his support and apprehension, and also for his guidance. I have learned a lot from him.

I also want to thank Aslı Selay Darğa for her understanding and love which made life easier for me and made this work be completed.

I would like to thank my housemates Korhan Sezgiker, İrem Vural and Tuğçe Yiğit for their understanding and help.

I am grateful to my lovely family for their support, love and encouragement through all my life.

Last, but not least, I thank the Scientific and Technological Research Council of Turkey (TUBITAK) for their financial support with grant code BIDEB2228 in the research that made it possible for me to conduct this thesis study.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF SYMBOLS

SYMBOLS

n	Number of pixels in one dimension of a square patch, each patch contains nxn pixels
m	Number of bins in one dimension of the joint histograms
k_0	Number of Gaussian models to be learned in the first frame of the algorithm
k	Number of Gaussian models to be learned in each cycle of the algorithm
K	The maximum size of the Gaussian model library
N	Minimum number of training samples required to learn k new Gaussian models
T_r	Roughness threshold
T_m	Gaussian model update threshold
T_c	Classification threshold
$\boldsymbol{\mu}$	Mean vector
$\boldsymbol{\Sigma}$	Covariance matrix
w	Weight of a Gaussian model
$\boldsymbol{M}$	Gaussian model
p	Image or point-cloud patch
$\boldsymbol{x}$	Feature vector of a patch
S	Total number of samples
G	Total number of groups or clusters

LIST OF ABBREVIATIONS

ABBREVIATIONS

CUDA	Compute Unified Device Architecture
GMM	Gaussian Mixture Models
RGB	Red-Green-Blue
HSV	Hue-Saturation-Value
LADAR	Laser Range Finder
DARPA	The Defense Advanced Research Projects Agency
SVM	Support Vector Machines
MoG	Mixture of Gaussians
CCA	Continuous Classification Accuracy
CG	Candidate Ground
CO	Candidate Obstacle
IMU	Inertial Measurement Unit
GPU	Graphical Processing Unit
OpenCV	Open-source Computer Vision Library
FPS	Frames per Second
GB	Gigabytes
RAM	Random Access Memory
CCD	Charge-coupled Device
METU	Middle East Technical University
UPS	Uninterrupted Power Supply

TP	True Positives
FP	False Positives
TN	True Negatives
FN	False Negatives
EM	Expectation-Maximization Algorithm
SST	Total Sum of Squares
SSW	Within-cluster Sum of Squares
OpenCL	Open Computing Language
SDK	Software Development Kit
CPU	Central Processing Unit
Cdf	Cumulative Distribution Function

CHAPTER 1

INTRODUCTION

1.1 Mobile Robotics

Robotics technology has been an active area of research for about half a century now and began to be used in manufacturing in early 70s. The main use of robotics technology in manufacturing was to replace the repetitive work done by human to decrease accidental risks and the costs with an increase in the production quality. The first applications of robotics products were in structured environments with predefined motion paths. However, as robotics technology develops the application areas were grown to dynamic environments without predefined paths. In order to deal with those applications the robots used become mobile and improved.

Figure 1 - An example of industrial robots; welding robot [1].

For the past decades, mobile robotics, which involves different scientific disciplines such as computer science, mechanical engineering, electrical engineering, cognitive science, materials

science, has been an active research area due to the need of autonomous mobile activities. Different than industrial robots or industrial manipulators, mobile robots have the capability of moving autonomously. This capability makes them popular in various different real life applications. While some of these applications are easy to solve, some others consist hazardous environments and require precise perception, judgment and actuation. They are usually expected to accomplish these tasks by navigating dynamically in those environments consisting humans, other robots, objects and obstacles.

Figure 2 - Some examples of mobile robots: (Left) Google's self-driving car [2]. (Right) Samsung Navibot robotic vacuum cleaner [3].

The examples of these applications can be given as car driving, autonomous flight, cleaning, space exploration, delivery, mine sweeping, aiding and rehabilitation. In each application space, the requirements are different and hence, each application has different problems and requires a variety of solutions to these problems. Although the scientific community extensively researched on the field, suggested a bunch of solutions to the different problems related with the area, and even some commercial applications are available, the research field is still intensely active and attractive for the researchers.

1.2 Road Detection

In order to accomplish a given mobile task successfully, the robot should be able to navigate autonomously. While navigating, the robot should be able to perceive its environment correctly, avoid hazardous situations and make appropriate decisions about the environment or actions.

Depending on the application a wrong implementation may yield catastrophic consequences for humans and environment besides the failure of the mission. The robot may even destroy itself due to a wrong perception, decision or action. Thus, to accomplish a successful navigation, road detection or road perception is a must to take the right action.

In general roads can be classified as structured and unstructured. Structured roads are the roads which have common structures such as highways or urban roads. They usually have lane markings, pavements and a common color. Their surfaces are smooth and hence they are more suitable for driving. The examples of such roads can be seen in Figure 3.

Figure 3 - Two examples of structured roads: (Left) Highway [4] and (Right) urban road [5].

Unstructured roads, on the other hand, do not have common structures such as pavements or driving lanes and usually are dangerous to navigate. In fact, there might not be a path or road but some traversable regions. Their surfaces are rough and the color of the road can vary as it is given in Figure 4.

Figure 4 - Two examples of unstructured roads [6], [7].

Since structured roads have common properties, the research on structured road detection problem led to useful results faster than the research on unstructured road detection. Thus some successful solutions do exist on structured road detection problem, but unstructured road detection is still an active research area.

1.3 Scope of the Thesis

In this thesis, the primary objective is to detect the traversable part of the terrain or the road region in unstructured environments by using a stereo camera with real-time processing. Other than the parameters required by the algorithm, the inputs used are the color image and point cloud data of the environment supplied by the camera.

The aim of the study is to develop an algorithm which;

- detects all kinds of traversable or road regions primarily on unstructured environments,
- is robust to changing light conditions,
- adapts itself quickly to the changing road conditions, types and appearances,
- can work in real-time.

1.4 Contributions of the Thesis

In this thesis work a GPU-accelerated adaptive unstructured road detection algorithm is proposed. The contribution of this study is many. Firstly, a simple thresholding based on height variance of road/non-road regions is proposed to gather training samples from road regions automatically. Most of the literature on adaptive unstructured road detection achieves this by

offline trained classifiers which require lots of labeled training samples. Using offline training makes algorithm structures complex, algorithm efficiencies low and introduces training time to run the algorithms. Secondly, the adverse effect of sudden illumination and light intensity changes on road detection algorithms is addressed. In addition to these, integration of an estimation procedure for selecting number of models to be learned from the gathered data is proposed. In order to select the best distance measure for classification, a comparison for six different distance metrics are given. Lastly, manycore parallelization of the algorithm is done and implemented using CUDA resulting processing speeds of 50 Hz in the worst case.

1.5 Outline of the Thesis

The outline of the study can be summarized as follows:

In Chapter 2, a literature survey on unstructured road detection algorithms are presented and concluded results are discussed.

In Chapter 3, data collection from the unstructured environment to develop the method is presented. The hardware and software environments used are given with the structure of the algorithm proposed. Experiments conducted and results on performance are presented with a discussion.

In Chapter 4, massively parallelization and CUDA implementation of the proposed algorithm are presented with the experiments and performance results.

In Chapter 5, which is the final, a conclusion is given and the future directions on research are suggested.

CHAPTER 2

LITERATURE SURVEY

2.1 Introduction

In this chapter, a survey on traversable road recognition, detection and following in unstructured or off-road terrains in the domain of mobile robotics or intelligent vehicles is presented. The unstructured roads or pathways, which can be found often in nature, do not have common structures or properties unlike urban roads or highways as it can be seen in Figure 5. In unstructured roads the road shape is usually arbitrary, the road surface can be degraded and road boundary can be unclear [8]. Furthermore, the road appearance or shape can change along the course. Thus, the problem of detecting and/or following road regions in unstructured environments is an ill posed problem and a variety of solutions are proposed in the literature. Some of these solutions use specific color and geometrical models, some of them are developed for specific types of roads, some solutions approach the problem as a machine learning problem and some approaches generate a more general solutions that can be applied to all type of roads and terrains which can also be adapted to the road. Although there are adaptive approaches, they have either too complex structures or are working slowly. The simpler adaptive approaches on the other hand require an offline training phase and hence the preparations for the use in mission take long times and require hard work. Thus, an adaptive unstructured road detection algorithm having a simple structure with low running times and without the need of offline training can be a more suitable alternative for the use.

Figure 5 - Unstructured roads in the pathways of Yalıncak Village in METU campus.

2.2 Literature on Unstructured Road Detection

When the literature on unstructured road detection problem is considered, a variety of methods have been proposed in the past few decades. Some of these methods use geometrical features, while some others use color and texture information. There are also couple of hybrid models are available in the literature. For generating the road model, some researchers apply simple geometrical, probabilistic or color models while some others approach the problem as a statistical machine learning problem [9-27]. There are also self-learning and adaptive approaches that update constructed models throughout the mission [28-51].

In their work Ekinci et al. [9] designed a road and junction detection and following algorithm around a constructed geometrical model. They modeled the road boundaries as two parallel curves via a second order polynomial and then the road was segmented using gray levels and texture representations obtained by Roberts operator. Afterwards the road model was updated in order to obtain the road boundaries exactly.

In the study done by Jansen et al. [10], the authors classified road regions via Gaussian Mixture Models (GMM) in RGB color space. According to their assumption, the regions of similar environments with similar geometry (height, roughness and shape) have similar color distributions (appearance). Using this assumption, regions in an image such as sand, sky, foliage and etc. are modeled via GMMs using labeled sets and these GMMs are used then for terrain classification.

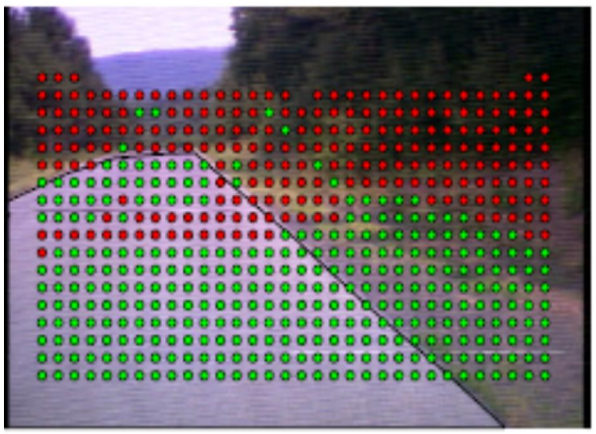

Figure 6 - Classification result of the algorithm developed in [11] by Rasmussen.

In his work Rasmussen [11], used the combination of features coming both from color camera and laser range sensor for unstructured road detection. Joint histograms of image patches used as color features while texture features are obtained by applying Gabor filters to these patches. Geometrical features were calculated from the laser range measurements. All the features of training images then fed to a two layer neural network for training. Afterwards, this trained net used as road/non-road classifier. One of the most elegant geometrical features used in this work is the height variance of field areas since height variance is statistically sound and robust to noise. In another studies done by the researcher [12], [13], Rasmussen utilized Gabor filters for extracting texture features from the image. Using a voting procedure, the position of the vanishing point, the point where the road boundaries are intersected on the horizon, was found. Then this point in consecutive frames used to estimate the road boundaries by the algorithm.

In their study, Zhang and Kleeman [14] modeled the roads via rapidly adapting 3D color histograms and geometric constraints. The road probability image constructed utilizing the image data coming from a panoramic camera. To this probability image, a road model was tried to fit and this model then used to detect road regions in the image.

Cheng et al. [15] represented an approach that can be applied both structured and unstructured roads. In this approach color features of road images were used and then mean shift algorithm was applied to cluster feature vectors into road/non-road regions.

Hu et al. [16] used average color values and standard deviations as color features, edges extracted via canny edge detector as edge features and road width as the shape feature. Then by using these features, pixels were clustered into road, non-road and ambiguous regions.

In their work Huang et al. [17] used HSV color space and different road properties to generate feature vectors. Color features are calculated from saturation and value channels of HSV color space. The road boundaries assumed to be parallel and misclassifications were eliminated using this assumption.

In their study Gao et al. [18] used features obtained both from RGB and HSV color spaces. The features used were the average value of each channel in the image patch.

In a study carried out by Wang et al. [8], color, edge and texture features were used for classification and learning. Color features were normalized average color channel values, edge features were the edge image and texture features were contrast, energy and correlation properties of the image. Then, Support vector machines were used as the road classifier.

Alon et al. [19] used Oriented Gaussian Derivative Filters, Walsh-Hadamard kernels and Moments method for texture feature extraction. Then using Adaboost algorithm the image was segmented into road and non-road regions. Then, these results are combined with geometric projection to recover Pitch and Yaw.

In [20], Tan et al. used a monocular color camera as sensor and assumed the bottom part of the camera images showed the road surface for model training. In their approach they made use of color histograms in the normalized RG color space to model the road. To represent road, they used 4 models and to represent the background, they used one. These models updated dynamically frame by frame. The background model of the previous frame used to build the background model in the next frame. This way a temporal constraint was introduced for the background model.

The study carried out by [21] Zheng et al. assumed the road has a trapezoidal shape and selected a rectangular region in front of the vehicle guaranteed to be road. Then, HSV color space values of this rectangular region recorded and average color channel values of these recorded data calculated. Using color filtering of these values on the image and geometrical constraints yielded road area in the image.

In a study held by Foedisch and Takeuchi [22], an adaptive road detection algorithm that uses color features derived from color histogram as features and artificial neural networks as classifier. Six windows on an image are marked for the generation of training samples automatically. Three of these windows are to gather road training samples and three of them are for non-road training samples. Features extracted from these six windows and artificial neural networks are trained continuously using these features.

The study done on the issue by Sun et al. [23] utilizes stereo vision for road detection. In order to generate training samples, a ground projected feature map which consists of the features obtained from the images projected to the ground terrain. Then a classifier is trained to separate road and non-road regions.

In [24], Lieb et al. proposed an algorithm that assumes the starting place of the vehicle as road and use this assumption as a template to find the road region in the successive frames with the help of reverse optical flow. Previous views of the road surface were computed using reverse optical flow, and then road appearance templates were learned for several target distances. The main drawback is that the use of optical flow estimation does not work well on chaotic roads, when the camera is unstable and the optical flow estimation is not robust enough.

Another road recognition algorithm is proposed by Li et al. [25] that can be used on both structured and unstructured roads. The algorithm uses fuzzy reasoning to extract edges and road regions are determined according to those edges.

Different than the usual approach Poppinga et al. [26] use an infrared camera and a stereo camera as sensors to obtain 3D data from the environment. Hough transform is employed to this 3D data and if the result does not resemble to a shape defined previously, this region is classified as road.

Wei and Gong [27] proposed an algorithm that utilizes Otsu's multi-threshold algorithm and two-peak algorithm to calculate road probabilities of the regions and to cluster the image. Also, Canny operator is applied to the image to extract road boundaries.

2.2.1 Near-to-Far Learning

Since the underlying road model in the unstructured outdoor environments is highly complicated, instead of using one or more static models, the studies are directed to techniques that uses adaptive or continuously learning multi-models. One such technique is called Near-to-far learning. Near-to-far learning technique is a recently proposed continuous learning method that uses a close-range (e.g. laser range finder) and a far-range sensor (e.g. camera) together in relation to detect unstructured roads autonomously. This method uses reliable close range sensor data to generate training samples automatically and continuously online. The training samples obtained from the close-range sensor is then used to learn a correspondence between the overlapped regions of close-range and far-range sensor data. This correspondence is then used to classify the traversable or road region in the far-range sensor data.

The work done on near-to-far learning in the domain of road detection can be classified according to the sensors used as stereo camera and laser range finder – camera combination.

2.2.1.1 Laser Range Finder – Camera Combination

The algorithms based on laser range finder – camera combination use a laser range finder for close-range sensor while using a monocular camera for the far-range sensor.

One of the first and the most important applications of the technique are done by Dahlkamp et al. [28] for the Stanford racing team on DARPA Grand Challenge 2005. The researchers proposed a self-supervised road detection algorithm that fuses laser data with monocular camera images for long distance adaptive road detection. The laser was used to find drivable regions and obstacles in near vicinity of the vehicle and hence generating the training data as described in [29]. The road regions found in the near vicinity is used as training data to generate mixture of Gaussians to learn an RGB model of the road and find the road from the monocular camera image. Although this approach does not require finding the non-road or obstacle samples as well as road samples, the authors used the road and the obstacle regions for global map representation. Later Nefian and Bradski [30] applied hierarchical Bayesian network for segmenting desert images and detecting off-road drivable corridors. They fused model geometrical and smoothness constraints such as road boundary and horizon semantics to the algorithm developed by Dahlkamp et al. [28] via a Bayesian image model.

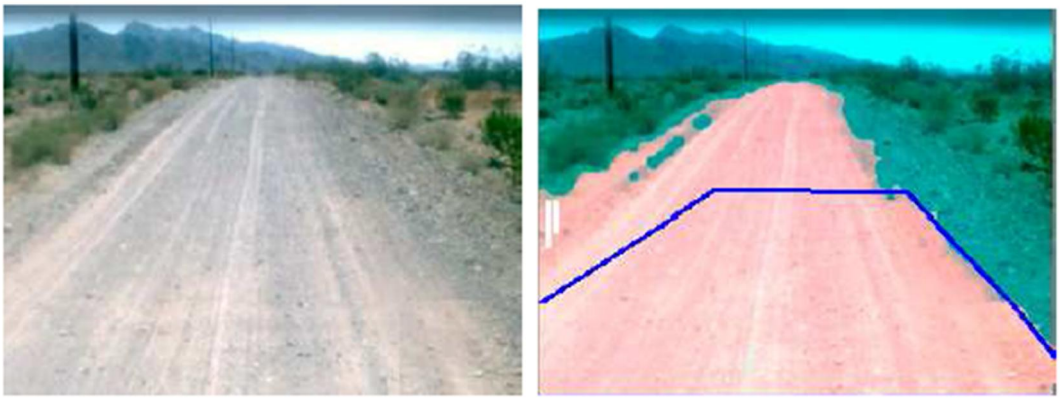

Figure 7 - Raw image and the classification result of the algorithm developed by Dahlkamp et al. [28].

In [31] the authors proposed a self-supervised terrain cost estimator algorithm by the utilization of the features from both aerial imagery and the robot's on-board sensors with the use of Bayesian scope learning model. An online probabilistic model was trained on satellite imagery and LADAR sensor data for the vehicle's navigation system. They applied a learning approach to integrate both general feature-based estimation and self-supervised locale-specific estimation to improve navigation capabilities for unmanned ground vehicles. In their application, they integrated overhead data and short-range sensor data into an online learning framework. Both local information and global information were used to yield a cost map to improve robot navigation. Experiments showed dramatic improvement of navigation performance in traversal time, distance traveled, and average speed.

Suzuki et al. [32] proposed techniques cope robustly with near-range road estimation using a laser scanner and long-range terrain classification using a color camera. Near-range road surface conditions were estimated by using information of remission values as reflectivity of a laser by modeling with a mixture of Gaussians. They applied graph cut algorithm to grid map in order to estimate road region robustly also in complex environments where fallen leaves exist sparsely. Moreover, the authors proposed superpixel-based terrain classification method which can give a good performance compared with pixel-based classification.

2.2.1.2 Stereo Camera

Most of the research done for autonomous unstructured road detection or traversability detection and terrain classification utilizing near-to-far learning use stereo vision as the primary sensor. The main reason behind it that the geometry data or point cloud data and color data comes registered, thus instead of utilizing two different sensors, one sensor is utilized as both the close-range and the far-range sensors.

Hadsell et al. [33] proposed a near-to-far self-supervised traversable road and obstacle detection algorithm for natural terrains that uses stereo vision for the main sensor. Due to the range limitation of stereo vision depth measurements (reliable up to 10-12 meters), the authors adopted a near-to-far self-supervised architecture. They used a stereo algorithm to produce a 3D point cloud; then ground plane and footline estimation methods are applied to separate these points into ground, obstacle, and footline classes. The authors used offline learned feature extractors which learned from an image patch database with distinct features [34]. The features were fed through the learning algorithm to train the online classifier [35], [36], [37]. The trained classifier were used to predict long-range visual data from images. The same team in another study [38] proposed a low cost robotic platform with no hand-tuned calibration and a solution to the "fast learning – fast forgetting" problem that arises in the long-term runs and degrades the performance of the system. The solution proposed for the memory is the use of multiple experts classifiers that can select the learned model appropriate to the environment.

Figure 8 - Raw image, traversability image from point cloud and final classification result of the algorithm proposed in [33].

Matthias et al. [39] used near-to-far learning approach that utilized both appearance and stereo information from the near field as inputs for training appearance-based models and then applied these models in the far field in order to predict safe terrain and obstacles farther out from the robot where stereo readings are unavailable (here, greater than 10 m). With such terrain predictions in the far field, the robot would follow a more natural path toward the goal, in this case avoiding trajectories toward the far-field obstacles. Terrain was represented as a collection of voxels. A voxel is simply a volume element. It can be thought as the volumetric version of a pixel. Each voxel had a density calculated from the number of range measurements obtained from that voxel. By using this density values as features for voxels of traversable and non-traversable, an SVM classifier was trained offline for close range-traversability. For the mid-field and far-field, three types of clustering methods were experimented to estimate the traversibility from the normalized RGB distribution of each cell: Unsupervised kMeans,

supervised kMeans and supervised MoG, where supervision was done by estimating a priori values for traversability using weak range measurements in the midrange.

In [40] and [41], Procopio et al. proposed the adaptation for selecting and combining models from an existing model library to the time-evolving data associated with the outdoor robot navigation domain. They showed an ensemble of models outperformed a single model fitted to the data. An ensemble can be constructed in a variety of ways; for example, it can be dynamically created over time in response to the incoming data stream. Alternatively, a library of one or more models may already exist in memory, and ensembles can be selected from this library to optimize performance on the current test data. These two approaches can also be combined; in this case, a library of models is available but is also modified on-line by adding new models over time (and, if appropriate, pruning irrelevant models from the library). It was shown that without prior models, the performance is better but the system starts without knowing anything and hence the risk is increased. The Ensemble Selection procedure was explained as follows:

- Start with the empty ensemble.
- Add to the ensemble the model in the library that maximizes the ensemble's performance to the mean CCA metric on a validation set of near-field stereo labels taken from the current image.
- Repeat Step 3 until one of the stopping criteria has been met.
- Apply each model in the resulting ensemble to the remainder of the image.
- Combine each model's output to obtain a final terrain classification and repeat Step 1.

Linear SVM was used for base learner in the paper for all models in the model library. In [42], Procopio et al. used L2-regularized logistic regression instead of linear SVM in order to decrease the computational time required for the training without loss of accuracy. In another study [43], the authors proposed a method to cope with the imbalanced training data usual in autonomous navigation domain. In [44], however, instead of using the linear combinations of models in the ensemble, the authors proposed the use of model abstinence since the inclusion of inappropriate models to prediction of ground or obstacle degrades the estimation accuracy. Thus, in the determination of where models are applicable, they used the distribution of training data as the source of the mixing coefficients that form the soft gating network in the mixture of experts model. This gating network determines which models are applicable and to what extent. Grudic and Mulligan [45] also proposed a new distance metric for road segmentation as polynomial Mahalanobis distance. The algorithm proposed in this paper used color and texture properties of a reference road region as a seed for segmentation.

In [46], the authors proposed a near-to-far learning method that uses stereo vision as primary sensor. Images, which were obtained by a stereo camera, used for geometrical and appearance based traversability analysis. Stereo matching and noise removal were done by acquired images. Next, plane was estimated from the point cloud. Finally, geometrical traversability was calculated. Then, features were acquired from images. In this research, Hue and Saturation of

16×16 2D histogram were used for learning each 33×33 image patches and classified by SVM. After that, predictive probability was calculated. Finally, appearance traversability was analyzed from probability of each cell of polar map. Final traversability was determined by two results of traversability; one is coming from the geometry and the other coming from the appearance. In this stage, probability of classification was used for reducing the effects of false positives and negatives.

The use of stereo vision with near-to-far learning was also used by [47], where the algorithm involves segmenting the far field region of an image frame into a collection of segments. These were combined with ground/obstacle pixel labels from the robot's stereo system. Segments which have any overlap with ground segments and segments immediately neighboring such segments made up the set Candidate Ground and those which overlap obstacle segments and segments immediately neighboring them make up Candidate Obstacle. If a segment overlaps (or neighbors) both ground and obstacle labels then it is deemed ambiguous and belongs to the set Candidate Ambiguous. All segments in CG which are closer to ground segments than some threshold "dg" and those in CO that are closer to obstacle segments than some threshold "do" in some feature space f, according to some similarity measure D, are labeled as "Ground plane" and "Obstacle" respectively. Finally, segments in CG and CO which are closer to both the stereo ground plane and the stereo obstacle regions than their respective thresholds were deemed ambiguous and were classified as unknown. As color models, an RGB segment of 30 bins was used for each segment. Each class was modeled as a mixture of histograms of contiguous regions made up of the pixels belonging to that segment set S.

The authors of [48], proposed an algorithm for the classification of terrain traversability. A neural network was trained offline with hand-labeled 8 geometrical features that defines traversability and then this geometrical knowledge was associated with color information continuously by online unsupervised learning with a non-parametric model. The near-to-far approach used both appearance and stereo information from the near field as inputs for training appearance-based models and then applied these models in the far field in order to predict safe terrain and obstacles farther out from the robot where stereo readings are unavailable (here, greater than 10 m).

Kim et al. [49], proposed a self-supervised traversability estimator algorithm in which the robot learns the affordance of traversability through the interactions with the environment without human intervention. A correspondence was established between the experience of the robot and the visual features from the stereo cameras by clustering. The experience of the robot here is the IMU, motor current and bumper switch on the robot. A 13 dimensional feature vector composed of 5 bins height histogram and 8 dimensional maximum Laws texture mask used for the image patch representation. Region primitives, such as patches, were preferred since they allow the robot to estimate the characteristics of the image region using rich features such as color and texture distributions. However, fixed-sized patches lack the ability to correctly identify the boundaries of complex objects, with the result that multiple distinct image regions may be contained within each patch, degrading the estimation accuracy.

Angelova et al. [50], developed an algorithm that uses the slip of the vehicle as the supervisor to reduce the dimensionality of the visual data and classify the terrain with probabilistic models. Slip is learned in a Mixture of Experts framework: terrain was classified first using appearance information and then slip, as a function of terrain slopes, was learned. The rationale for doing that was: 1) terrain type and appearance were approximately independent of slope; 2) introducing this structure helped constrain learning to better balance limited training data and a potentially large set of texture features.

Konolige et al. [51] used geometrical and color information to find road area using stereo vision. They combined vision and depth into one system, to mitigate the drawbacks of each approach. They first estimated a ground plane from the disparity image and then established a sight line on the ground plane. By using this sight line, they established a Gaussian model of normalized colors and classify the region according to that model. If the classified road region is geometrically path-like, it is used as an input to the learning algorithm.

In conclusion, near-to-far learning technique has promising results due to its adaptive and self-training nature. The techniques that do not utilize near-to-far learning scheme either cannot deal with changing environment and road conditions or can be applied on a specific environment. The technique seems strong when combined with a stereo camera since stereo camera provides both reliable close-range information and rich far-field data. One main disadvantage of the method is its complex structure especially in near-field training sample generation. The studies in the literature deal it with offline trained classifier which requires labeled training data and hence a lot of unqualified hard work. Offline training also degrades the generality of the algorithm. The research proposed in this study approaches this problem with a simple thresholding and hence decreases the structure complexity and increases the generalization. In this study also the issues related to light changes are addressed. According to the survey done, it is also found out that that none of the described solutions implemented GPU acceleration. In this respect, this study is a pioneer.

CHAPTER 3

PROPOSED ALGORITHM

3.1 Introduction

In this chapter, the proposed algorithm for adaptive unstructured road detection is explained in detail. For some parts of the algorithm several approaches are considered and evaluated in order to select the best alternatives. The algorithm proposed in this study is based on the Gaussian model learning and updating scheme proposed by Dahlkamp et al. in [28], though, there are a number of major and minor changes done. One of the most important differences is using a stereo camera both for close-range and far-range sensor instead of using a LADAR for close-range and a monocular camera for the far-range. The use of stereo camera removes the need for registration between close-range and far-range sensor and as a result removes the registration errors and the time required for registration. The second major difference is the use of HSV color space instead of RGB color space. HSV color space represents the chromatic information and brightness information in separate channels. So in order to decrease the illumination effects, hue and saturation channels are used for color information. As feature vectors the joint histograms of hue and saturation channels are used to model the road appearance. As another major difference, the automatic training sample gathering procedure is simplified. The authors of [28] used offline trained Hidden Markov Models for training sample gathering. Since this training requires labeled data and hard work, it is replaced with a simple thresholding procedure using height variances of road regions. This removes the need for offline training and hence labeled training data. With this simple thresholding, the preparation of the system for running decreases significantly. Within the context of this study, agglomerative hierarchical clustering is applied to the training samples gathered in each frame to estimate the number of clusters residing in the samples and using this estimation Expectation algorithm is run. The effect of using hierarchical clustering for the estimation of number of models to be learned on performance is analyzed. The effect of six different distance metrics on classification performance is also analyzed.

Throughout the chapter firstly, the hardware and software environments used are described. After that, the data gathered for the development and evaluation of the algorithm and how these data are gathered are explained. Then in the main body of the chapter, the algorithm structure is explained and complexity analysis is given in detail. After that, the experiments done are explained and results are given with a discussion. At last, conclusions related to developed algorithm are drawn.

3.2 Hardware and Software Environment

Throughout the development of the algorithm all implementation is done on a Windows 7 64-bit machine using C++ and Visual Studio 2008 [52]. Most of the body of the algorithm is developed using OpenCV's [53] data structures, image processing and machine learning algorithms when applicable. Whenever OpenCV's native algorithms are not enough, custom C++ code is developed.

Evaluation and ground truth labeling software is developed using Visual Studio C# 2010 with C# programming language and AForge.NET [54] image processing, artificial intelligence and robotics library.

In order to gather stereo data Bumblebee®2 Stereovision Camera System by PointGrey Research [55] and its libraries are used. The stereovision camera has two CCD cameras each of which has the capability of delivering images of 640x480 resolution at 48 fps and 1024x768 resolution at 20 fps. However, the stereo data speed coming from the data stream is about 15 fps for image size of 640x480 on an Intel Core i7 Windows 7 64-bit system with 8 GBs of RAM. The two cameras has an offset of 120 mm between them. The camera libraries have the ability to supply the image from two cameras and the registered point cloud data to these images. Figure 9 shows sample left and right images taken from the camera.

Figure 9 - Left and right images taken from Bumblebee2 stereovision camera.

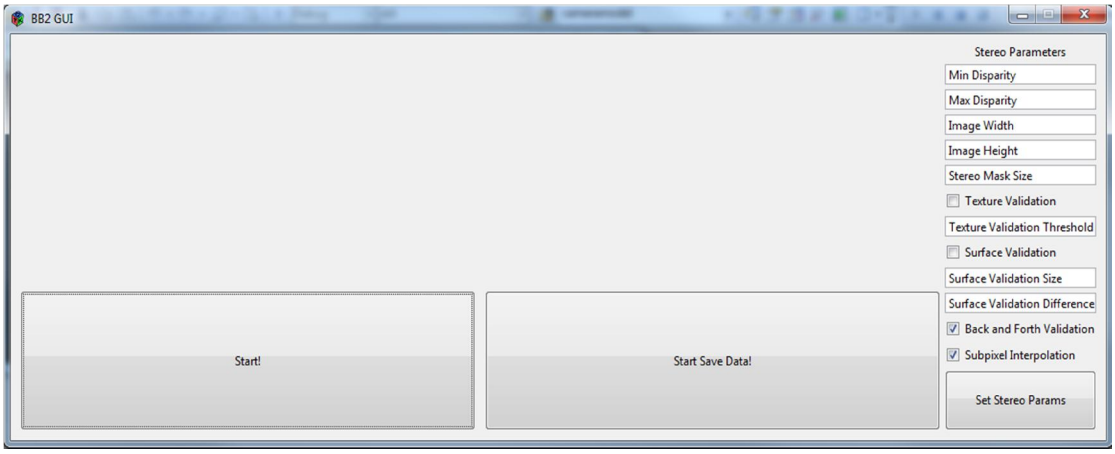

Figure 10 - Data collection software for Bumblebee2 camera.

3.3 Data Gathering

In order to develop the algorithm and run the experiments to test the success of the algorithm designed, color images and point cloud data of unstructured scenes are gathered. Data is collected from the pathways of Yalincak village in METU campus. A data collection software is developed using Bumblebee2 stereovision camera system's native libraries and visual studio c++ 2008 which can be seen in Figure 10. The software is run with Bumblebee2 stereovision camera connected to a laptop supplied with a UPS and all data is gathered while the camera is held by hand through a course of about 3 km long. The data is gathered between mid to late afternoon in September through a time window of 50 minutes. Since the camera is held by hand and white balance is active, small tilt and pitch of the camera results in significant light intensity changes from frame to frame. This makes the dataset suitable for the development and evaluation of an algorithm that is robust to illumination changes. Both stereo and color images gathered has a resolution of 640x480 pixels and a total of 360 images are obtained. Data collection course obtained from Google Maps [56] can be seen as an aerial image in Figure 11.

3.4 Algorithm Structure

A stereovision camera system supplies two kinds of data to the user. First one is the images and the other one is the point cloud data of the environment. According to the type of the stereovision system, there can be two or more color or monochromatic cameras on the system. In the vision system used in this work, there are two color cameras thus two color images coming through the sensor stream. Point cloud information is actually derived from these images by triangulation as described in [57]. As stated in the literature [33], [48] there are two main drawbacks of using a stereovision system. First problem is that as the distance from camera grows, the point cloud data becomes sparse. The reason of this sparsity is the covered area grow

with the distance while the resolution of the camera is constant. The second issue is also related with the first one. As the distance from the camera grows, the error of the measurements increase with the square of the distance from the camera as visualized in Figure 12 and Figure 13.

Figure 11 - Aerial image of data collection course obtained from Google Maps [56].

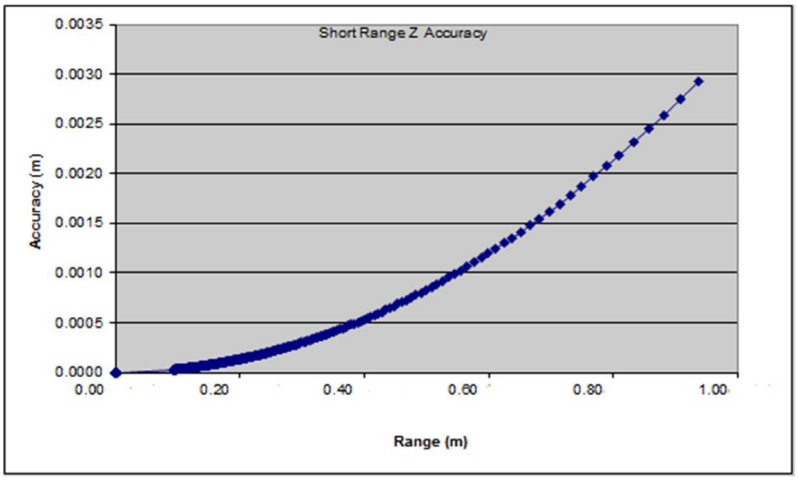

Figure 12 - Short range error in stereovision system [58].

As a result of this behavior of the stereovision system, the point cloud data coming from the stereo stream becomes unusable for a reliable autonomous navigation after approximately 10 m of distance from the camera. For this reason the designed algorithm finds smooth regions in the close range environment from point cloud and uses their color distribution to learn new road appearance models and to update older ones if any. Learned models are then used to classify road regions in the whole image.

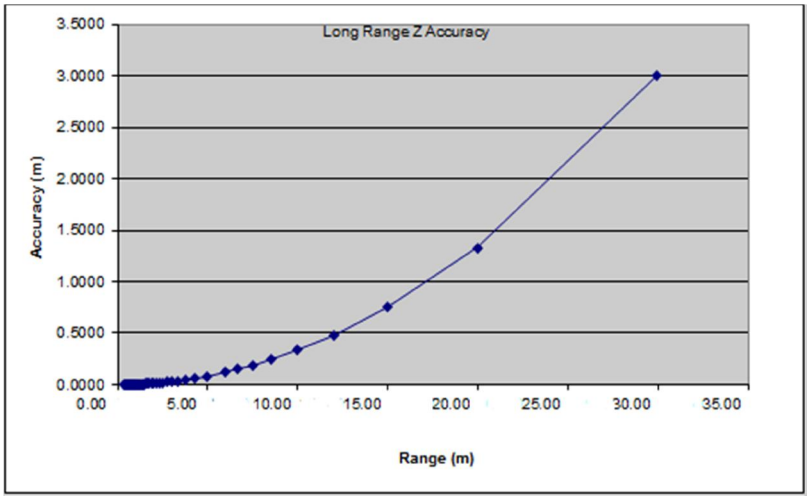

Figure 13 - Long range error in stereovision system [58].

The adaptive unstructured road detection algorithm developed in this study can be divided into five main subparts:

- Data preprocessing
- Feature extraction
- Automatic training sample generation
- Color distribution model learning and updating
- Classification

After the assumptions done in the design of the algorithm is given, each of the subparts are explained in detail in the following subsections. For an overview the flow chart of the algorithm is given in Figure 14.

3.4.1 Assumptions

The design of the algorithm explained in this study is based on three main assumptions. Firstly it is assumed that, ground height variation, variance or difference is high in the areas where the terrain is uneven or rough. When one looks at the examples of unstructured environments, it is easy to see that this assumption holds most of the time. Several examples can be given as in Figure 15.

The second assumption is that the color distribution in road appearance is different than the color distribution of the non-road regions. Although this assumption seems like a strong assumption, it causes troubles especially features spaces of low dimensions. As the dimension of feature space used increases, the dimension holds true most of the time. However, it is still weak if there is a camouflaged obstacle on the road.

The last assumption is that the camera is assumed to be parallel to the ground. The algorithm proposed in this paper uses one of the camera coordinates as height and hence too much deviation from this assumption will affect the algorithm's performance significantly. However, as the experiments suggest, small deviations does not affect the performance. In the future, the angle between camera and the ground can be measured and easily integrated to the system to remove this assumption.

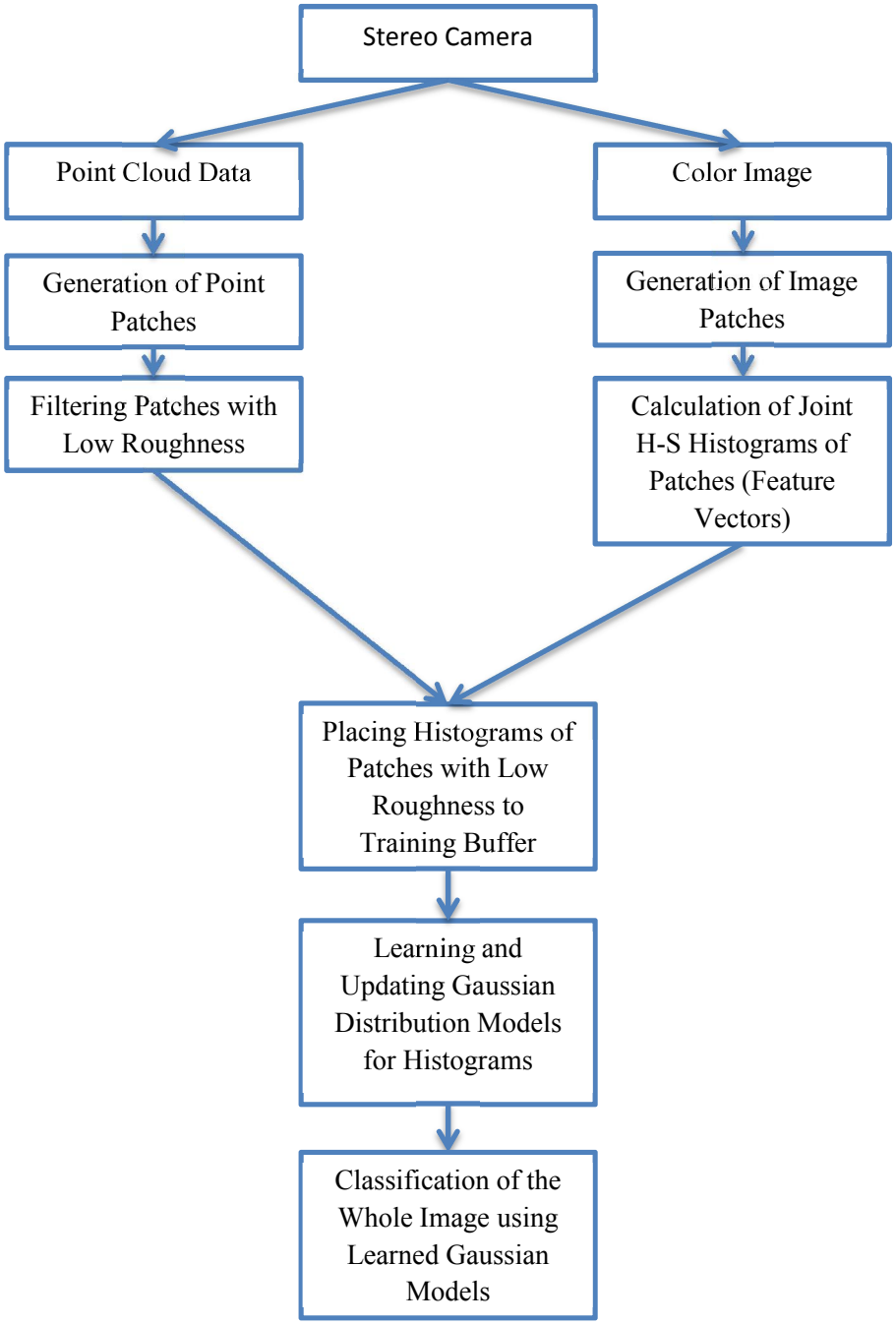

Figure 14 - Overview of the proposed algorithm for each frame coming from the camera.

Figure 15 - Unstructured Road Examples.

Figure 16 - Raw images taken from the camera (top row). Images after histogram equalization (bottom row).

3.4.2 Data Preprocessing

As mentioned at the beginning of the section, stereovision system supplies both point cloud data and color images. In order to extract color distribution, decrease the effect of noise and sharpen the color information, these raw data are preprocessed. In the first step, color image is processed with the utilization of histogram equalization to sharpen the constrast and increase color differences and then the image is converted from RGB color space to HSV color space. In the second step, both point cloud data and image are divided into patches to prepare for the histogram calculation and to increase the processing speed with subsampling.

3.4.3.1 Histogram Equalization and RGB-HSV Conversion

The first step of the preprocessing is the application of histogram equalization on each color channel of the color image supplied by the camera. The reason of applying histogram equalization on the RGB image is to reduce the adverse effects of unbalanced illumination and increase the contrast in the image so the chromatic content of the image becomes more distinguishable and uniformly distributed. Some images taken from the field and their appearance after the application of histogram equalization can be seen in Figure 16.

Table 1- Classification results of the proposed algorithm for two different sets of algorithm parameters with histogram equalization and without histogram equalization.

		without Hist. Equalization				**with Hist. Equalization**			
		TP %	**TN %**	**FP %**	**FN %**	**TP %**	**TN %**	**FP %**	**FN %**
Config 1	**Ave %**	81.99	87.50	12.50	18.01	69.84	96.04	3.96	30.16
	Std Dev	16.05	9.39	9.39	16.05	18.42	3.20	3.20	18.42
Config 2	**Ave %**	89.21	76.25	23.75	10.79	83.98	90.38	9.62	16.02
	Std Dev	14.78	13.68	13.68	14.78	14.58	6.77	6.77	14.58

Application of histogram equalization is especially useful for the regions in the image with low saturation and low value channel values. After converting RGB image to HSV, pixels with very low saturation and low value yields misclassification and more importantly increases false positive rates which can be fatal for an autonomous agent. However, the downside of using histogram equalization is the slight deformation of color content, thus the objects sometimes may not appear in their original colors. The other disadvantage is that the histogram equalization is done based on the content of each image while colors distribution models are generated and updated within different frames. This situation yields a decrease in the true positive rate (TP). Although histogram equalization damages the true positive rate (TP) success of the algorithm, since false positive rate (FP) decrease is much more than the decrease in true positive rate as can be seen from the experiments given in Table 1, histogram equalization is decided to be used. As Table 1 visualized, both the mean and standard deviation of false positive rate (FP) decreases dramatically. Since false positives (FP) yield risky judgements for the robot, their decrease is more important.

After the application of histogram equalization to the RGB image coming from the camera, next is the conversion of the RGB image to HSV color space. HSV color space is a cylindrical-coordinate representations of points in an RGB color model and can be visualised as given in Figure 17. Since HSV color space separates brightness information from chromatic information as a separate channel, it is more resistance to the light changes. Thus, only by using Hue and Saturation channels in the application, the system becomes more robust to light effects on the images.

The worst case time complexity for histogram equalization as well as RGB-HSV conversion for an image with I pixels is $O(I)$.

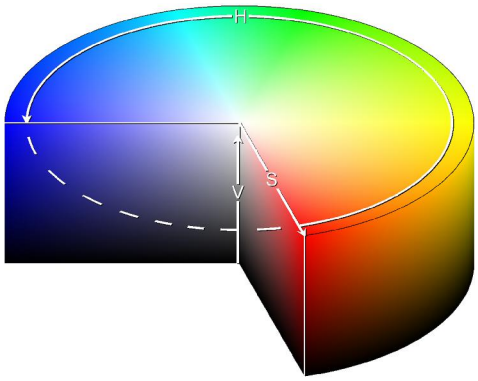

Figure 17 - HSV color space cylindrical coordinate representation [59].

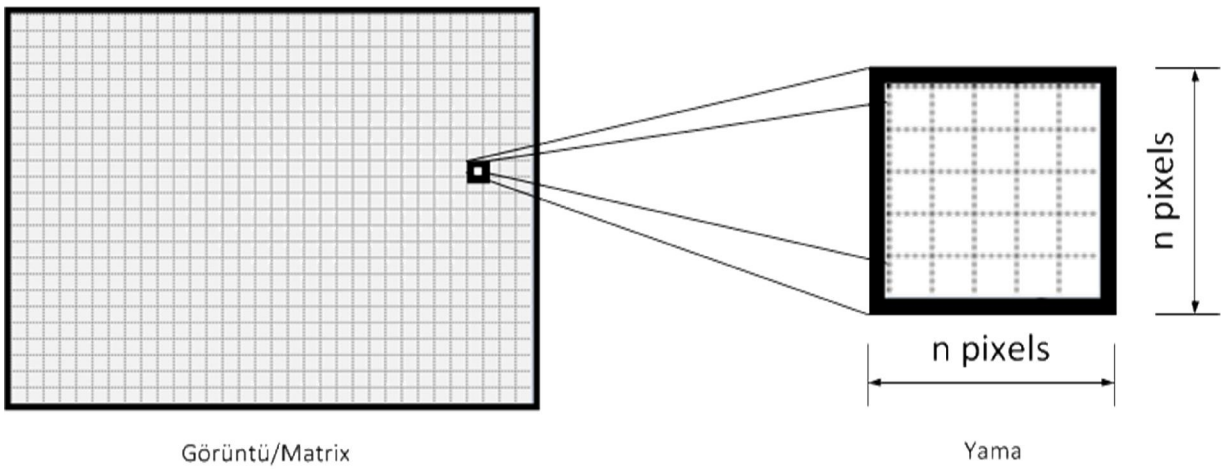

Figure 18 - Patch representation of data.

3.4.4.2 Generation of Patches

After conversion of the color image from RGB to HSV color space. Both point cloud data and color image are divided into patches of sizes n x n pixels. By dividing the data into patches, color distribution for image patches and height variations for point cloud patches can be obtained. Thus, the data become ready for feature extraction and since each region is represented with a patch, subsampling is achieved for increasing further processing speed. Patch representation can be visualized as given in Figure 18.

3.4.3 Feature Extraction

In order to represent or extract color and geometrical properties of the patches, it is required to calculate some features related to these patches. Since these features represent color and geometrical properties of the patches, extracted features are used to estimate or learn the road model and they are also used to classify whether a patch is belong to road or not. Usually extracted features are represented as a vector which is called feature vector. This feature vector is what represents the patch in the feature space where classification and learning is done.

3.4.3.1 Feature Extraction for Point Cloud Data

In designed algorithm, point cloud data is used to represent the roughness of the road regions. As it is mentioned at the beginning of the section, training samples that are used to learn color distribution models about the road are selected from the regions where ground roughness are low in the close-range. In order to find these regions, a feature that represents roughness should be used. In this study, the roughness feature used is the height variance within the patch. Although there can be other features used, height variance is statistically sound, robust to noise and as a result used in this study.

Height variance feature describes the deviation of the height within a point cloud patch. For a point cloud patch p of size n^2 pixels, this feature can be calculated by using (1)

$$Var_y(p) = \frac{1}{n^2-1}\sum_{i=1}^{n^2}(y_i - \mu)^2 \quad (1)$$

Where y is the y coordinate or height of the point according to the camera coordinate system and μ is the mean of the heights of the points in the patch.

The worst case time complexity for (1) for an image with I pixels is $O(I)$ with a single-pass variance calculation algorithm.

3.4.3.2 Feature Extraction for Color Data

In the algorithm proposed in this study, color distribution or appearance model of the road is the one wanted to learn. If the appearance of the road can be learned, then road and non-road regions can be classified using this information. In order to represent the color properties of a

region, joint histograms of hue and saturation channels of HSV color space is used. Due to the fact that value channel represents the brightness; it can be affected by the illumination easily. In order to obtain a road detection algorithm that is robust to light changes hue and saturation channel values are used. Joint histograms of these two channels are used as the feature vector to represent the color distribution of an image patch. Thus, for an m bin joint histogram, the feature space as well as the feature vector has m^2 dimensions. A schematic representation of H-S joint histogram is given in Figure 19 for better visualization.

The worst case time complexity for H-S joint histogram calculation for an image with I pixels is $O(I)$.

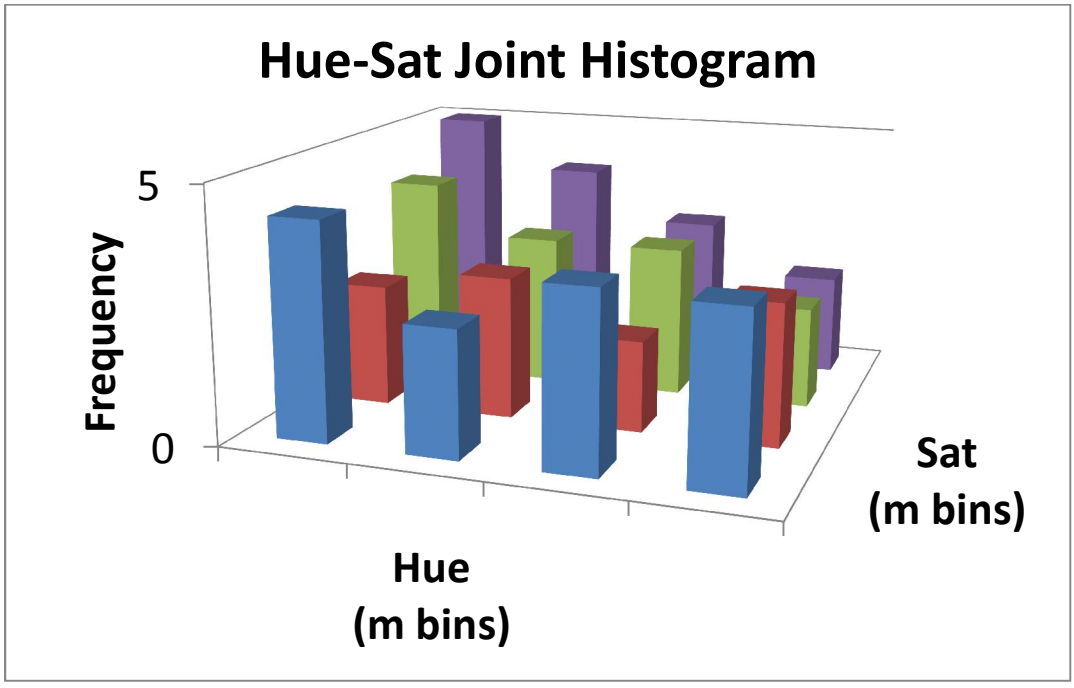

Figure 19 - Schematic representation of Hue-Saturation joint histogram of m bins.

3.4.4 Automatic Training Sample Generation

After extracting features for both point cloud data and image patches, next is to find the smooth enough regions in the environment. These smooth regions are filtered and later on their color distributions are used to learn color distribution models of road regions. In order to find the smooth enough regions to select as training samples, a roughness threshold should be defined.

This threshold is defined as T_r. Then all point cloud patches are filtered using this threshold. The patches which have a feature value smaller than T_r are accepted as training samples and those have a feature value greater than T_r are not accepted. For a patch p, this procedure can be represented as given in (2) for the proposed roughness feature.

$$\forall p \in \{Var_y(p) < T_r\} \tag{2}$$

After the application of (2), selected training sample patches on a single frame can be seen in Figure 20. If the number of patches are P, the application of (2) yields a worst case time complexity of $O(P)$.

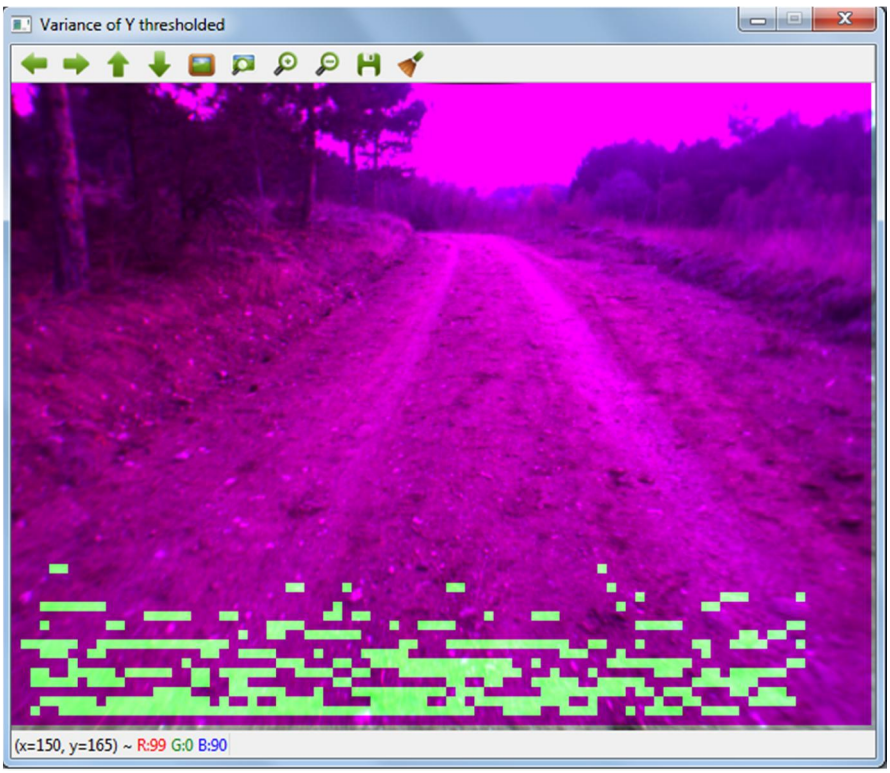

Figure 20 - Generated training samples after thresholding for a sample frame.

After roughness threshold is applied, in order to get the training samples with the highest quality and to be more robust against the noise, connected components labeling algorithm is applied. Connected components labeling algorithm is carried out as follows:

- A downsampled binary image is formed where each image pixel corresponds to an image patch of size n^2 pixels.
- In this downsampled binary image, the pixels corresponding to training sample patches selected in the thresholding phase marked as foreground and those not corresponding to training samples are marked as background.
- Connected components labeling is applied to this binary image and patch groups are labeled.
- Then the patches in the largest connected component are selected as final training samples.

After the application of connected components labeling and the selection of the patch group in the largest connected component from the data visualized in Figure 20, the resultant final training samples can be visualized in Figure 21.

If the number of patches are P and the number of components that will be found is L, the worst case time complexity of an efficient implementation of connected components labeling algorithm is $O(PL)$. Although the time complexity is high, as long as L is small, complMexity does not cause problems. According to the experiments conducted, L is rarely greater than 10, hence complexity does not matter much in the algorithm proposed.

3.4.5 Color Distribution Model Learning and Updating

After obtaining the training patches, next step is to construct color distribution models for road or updating the models learned in previous frames using freshly gathered training data. In this section color distribution model learning and updating scheme is explained. The learning and updating scheme explained here is proposed by Dahlkamp et al. in [28]. In this study it is used with minor adaptations and a hierarchical clustering scheme is integrated and its effect on performance is investigated.

3.4.5.1 Model Learning

After obtaining reliable training samples it is required to learn the appearance or color distribution models of road. Note that the training data are obtained only from road regions, not both road and non-road regions. Thus, the learning problem here is not the learning of a decision boundary that discriminates between two or more classes but learning the color distribution of road regions. The difference between model learning and decision boundary learning can be visualized with Figure 22. As mentioned in the previous sections, one possible shortcoming of this approach is that if a non-road region or patch has a similar color distribution with the training patches (a camouflaged obstacle), it can be classified as road. However, especially in high dimensional feature spaces the probability of coming across such a situation is very low. Thus it is assumed that non-road regions do not have similar color distributions with the road regions.

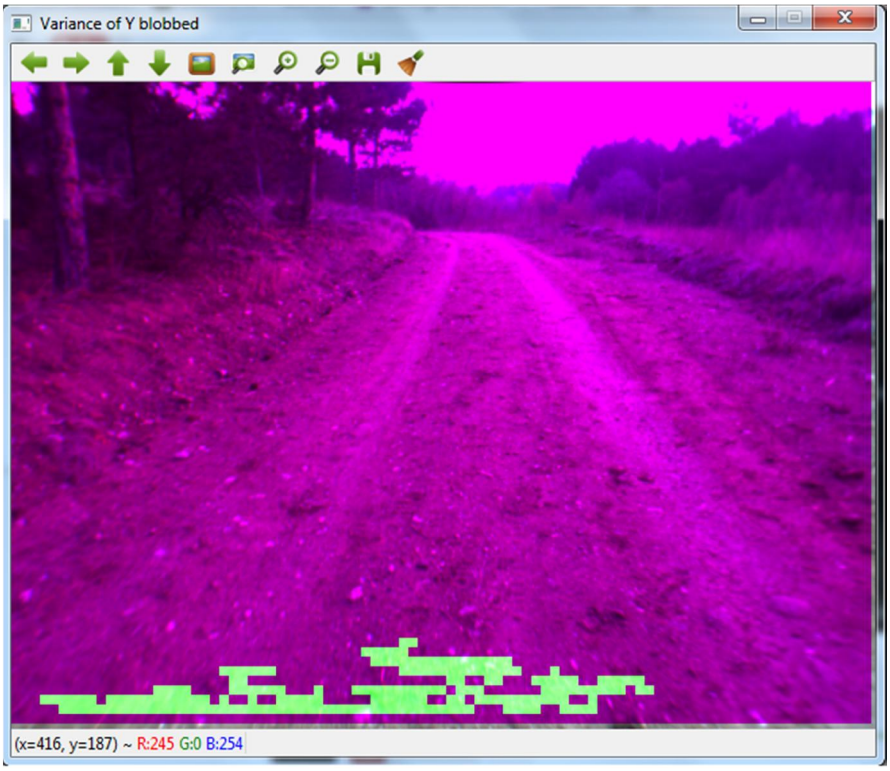

Figure 21 - Training samples obtained after the application of connected components labeling and selection of the largest connected component.

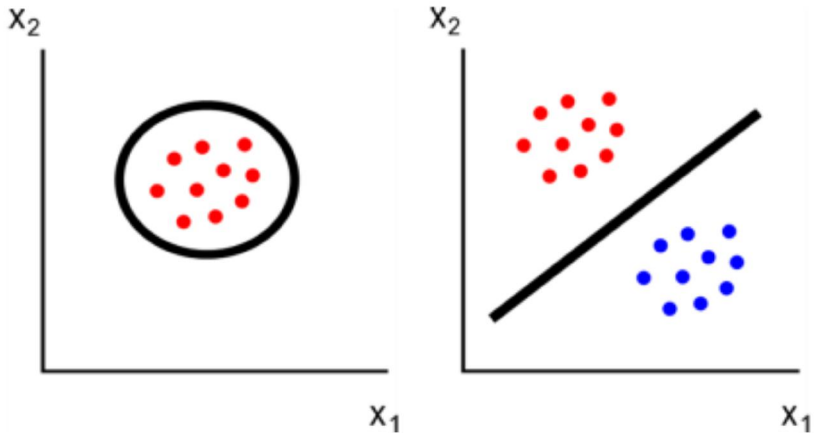

Figure 22 - Schematic for model learning (left) and decision boundary learning (right).

Since it is very hard (probably impossible) to model the road color distribution with only a single model, multiple Gaussian models are used in this study as well as in [28] to increase reliability, decrease the false classification rates without loss of generality and to adapt different road types and hence different road appearances as fast as possible. Since multiple Gaussian models are used to model the color distributions of road regions, it can be called as mixture of Gaussians.

One very appealing feature of Gaussians is that each Gaussian distribution model can be represented with three parameters, namely, mean vector ($\boldsymbol{\mu}$), covariance matrix ($\boldsymbol{\Sigma}$) and weight (w). If the dimension of H-S joint histogram of a patch is m^2, the mean vector ($\boldsymbol{\mu}$) becomes an m^2 dimensional vector, the covariance matrix ($\boldsymbol{\Sigma}$) becomes an $m^2 x m^2$ matrix and weight (w) is a scalar between 0 and 1 representing the number of training patches encapsulated by each Gaussian. In order to increase the speed of the algorithm and decrease the complexity, the covariance matrix ($\boldsymbol{\Sigma}$) is assumed to be a diagonal matrix.

In order to learn mixture of Gaussians from the training samples, Expectation Maximization (EM) [60], a well-known iterative clustering algorithm is used. In the application the maximum number of iterations of the algorithm is limited to 100. One issue with EM algorithm is that, it requires the number of Gaussian models to be learned as input, however, most of the time this information is not known. Thus, the number of models to be learned becomes a parameter of the algorithm. Another issue with the algorithm is the high time complexity. For an m^2 dimensional feature space as in this algorithm, k or k_0 clusters (Gaussian models) and c training samples, the time complexity of EM algorithm can be given as $O(c^{m^2k+1}logc)$ per iteration.

In the algorithm structure, each learned Gaussian distribution model is held in a model library. Due to performance limitations, this model library should have a maximum limit (K). When the system is first started, this library is empty and by using the training sample patches extracted from the first frame, k_0 Gaussians are developed with EM algorithm. Then, in each frame if there is enough training samples k new Gaussian models are learned and put in the model library. The reason of having two parameters as k_0 and k is that since there are no models in the library at the start of the system, the first learned models might have a bigger effect on the system success.

3.4.5.2 Model Updating

After the first frame, for each training patch obtained in each frame, Mahalanobis distance as given in (3), for that patch to each model in the model library is calculated.

$$dist_M(\boldsymbol{x_i}, \boldsymbol{M_j}) = \sqrt{(\boldsymbol{x_i} - \boldsymbol{\mu_j})^T \boldsymbol{\Sigma^{-1}} (\boldsymbol{x_i} - \boldsymbol{\mu_j})} \quad (3)$$

Where $\boldsymbol{x_i}$ is the i[th] training patch's feature vector and $\boldsymbol{M_j}$ is the j[th] Gaussian model in the model library. If the minimum Mahalanobis distance between a patch and the corresponding Gaussian

is smaller than model update threshold (T_m), this training sample patch is used to update that model's parameters using the following equations.

$$\boldsymbol{\mu_j} \leftarrow (w_j \boldsymbol{\mu_j} + w_i \boldsymbol{x_i})/(w_j + w_i) \tag{4}$$

$$\boldsymbol{\Sigma_j} \leftarrow (w_j \boldsymbol{\Sigma_j} + w_i \left((\boldsymbol{x_i} - \boldsymbol{\mu_j})(\boldsymbol{x_i} - \boldsymbol{\mu_j})^T \right))/(w_j + w_i) \tag{5}$$

$$w_j \leftarrow (w_j + w_i)/(1 + w_i) \tag{6}$$

In (4), (5) and (6) w_i is the weight of the training sample used to update the model and is the weight of $\boldsymbol{x_i}$ in all the data gathered in that frame. If the number of training samples obtained on that frame is S, it can be calculated simply as $1/S$.

If the number of training samples obtained in one frame is c, the worst case time complexity of finding the minimum Mahalanobis distance between a sample and the models in the library can be calculated as $O(cKm^6)$ where $O(m^6)$ term comes from Mahalanobis distance calculation in an m^2 dimensional feature space and K is the maximum number of models that can be stored in the model library.

If the number of training samples obtained in one frame is c, the worst case time complexity of (4), (5) and (6) becomes $O(cm^2)$, $O(cm^4)$ and $O(cK)$, respectively.

After updating, if there remains some training samples none of which are closer to any of the models in the model library than model update threshold (T_m), thus not used for model updates, two things can happen:

- If the number of these remaining samples is smaller than a minimum number N, these samples remain in the buffer and are passed to the next frame.
- Else, new k Gaussian models are estimated using EM and added to the model library.

When the model library reaches its maximum model number K, the k old models with the smallest weights are replaced by new learned k models. The summary of model learning and updating is given as a flow chart in Figure 23.

This kind of model memory scheme provides adaptation to both slowly and fast changing environments and prevents "fast-learn and fast-forget" problem stated in the literature [33].

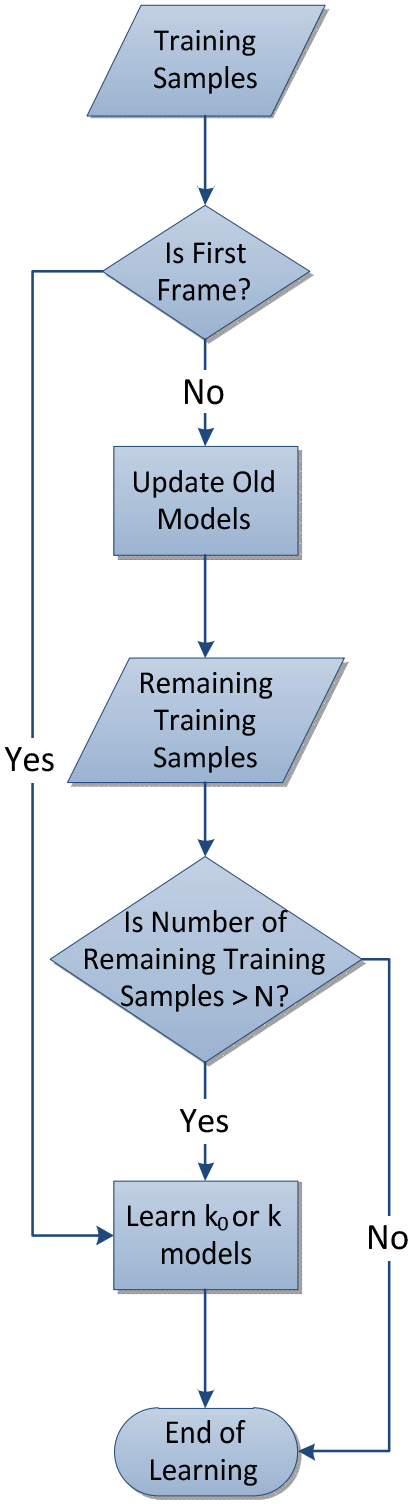

Figure 23 - Flow chart of learning & updating models.

3.4.5.3 Estimating the Number of Clusters in Training Samples

As mentioned in *Model Learning* section, EM algorithm requires the number of Gaussian models to be learned as input, however, most of the time this information is not known. Thus, the number of Gaussian models to be learned in the first frame (k_0) and in each frame (k) become two parameters to be selected. In order to remove the burden of selecting these parameters, despite of its algorithmic complexity agglomerative hierarchical clustering is applied to the training samples in each frame to estimate the number of natural clusters exist in the training set. After an estimate to the number of clusters is found out, this number is given as an input to EM algorithm to learn Gaussian models from the training set.

Hierarchical clustering is a method in clustering analysis that builds a hierarchy of clusters. Generally two types of hierarchical clustering are available: agglomerative and divisive.

In agglomerative hierarchical clustering, each sample in the dataset starts its own clusters. Thus, at the beginning the number of clusters is equal to the number of training samples. Then, in each step of the algorithm the clusters that are closest to each other according to some distance measure merged until there remains only one cluster that includes all training samples. Divisive hierarchical clustering is works just as the opposite of the agglomerative approach. A sample dendogram formed after the application of agglomerative hierarchical clustering is given in Figure 24.

In the literature, there is different distance measures used to calculate the distance between two clusters. However, none of them found superior to all of them. Depending on the application and dataset used, the success of the distance measure used differs. Three of mostly used distance measures can be listed as:

- Minimum distance (single-linkage or nearest neighbor)
- Maximum distance (complete-linkage or farthest neighbor)
- Average distance (mean-linkage)

In minimum distance measure, the Euclidian distance between the closest samples of clusters is taken as the distance between two clusters. For each samples a and b in clusters A and B, this measure can be expressed as in (7).

$$\min\{dist_E(a,b): a \in A, b \in B\} \quad (7)$$

In maximum distance measure, the Euclidian distance between the farthest samples of clusters is taken as the distance between two clusters. For each samples a and b in clusters A and B, this measure can be expressed as in (8).

$$\max\{dist_E(a,b): a \in A, b \in B\} \quad (8)$$

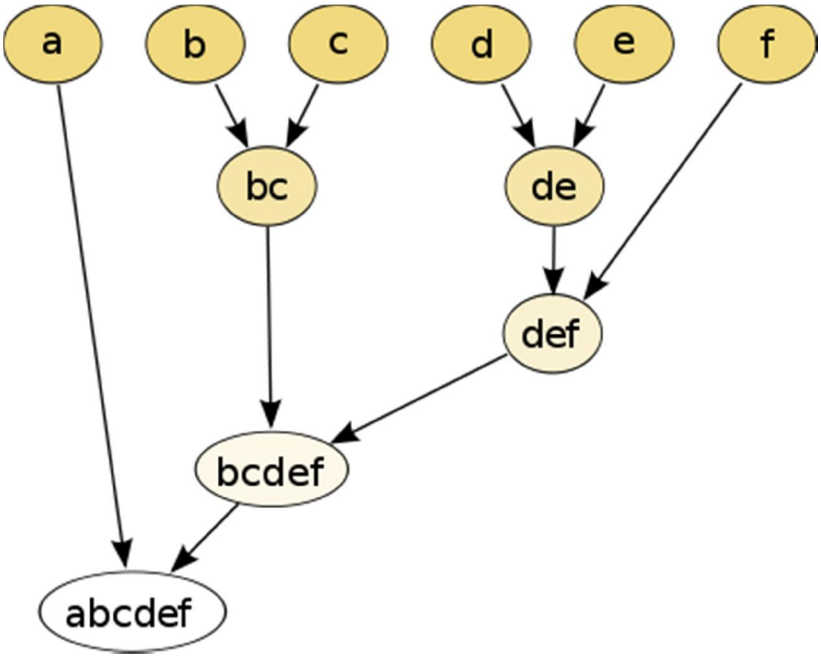

Figure 24 - Sample dendogram schematic formed after the application of agglomerative hierarchical clustering [61].

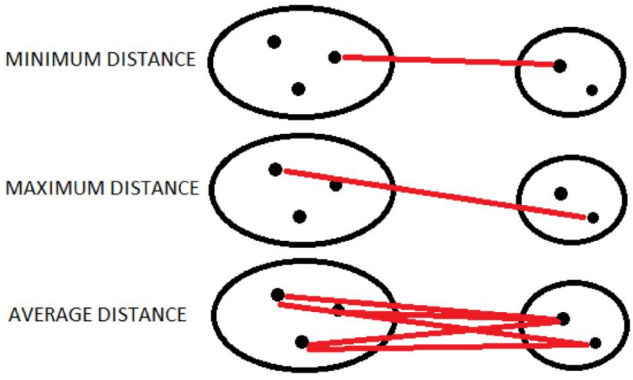

Figure 25 - Three distances between clusters: minimum, maximum and average distances.

In average distance measure, the average Euclidian distance between all samples of clusters is taken as the distance between two clusters. For each samples a and b in clusters A and B, this measure can be expressed as in (9).

$$\frac{1}{|A||B|}\sum_{a\in A}\sum_{b\in B} dist_E(a,b) \quad (9)$$

Where in (7), (8) and (9); $dist_E(a,b)$ represents the Euclidian distance between samples a and b taken from clusters A and B. A schematic for three distance metrics for better visualsization is given in Figure 25.

All distance metrics used in agglomerative hierarchical clustering result in different cluster shapes in the feature space. Use of minimum distance measure yields elongated or irregular shaped clusters with chopped distribution tails. Use of maximum distance measure, on the other hand, results in compact clusters with approximately equal diameters. Average distance measure is a compromise between minimum and maximum distance metrics and hence results in clusters shapes of neither elongated nor compact. Since, the real cluster shape is not known average distance measure is selected due to its compromised characteristics.

As it is mentioned at the beginning of the subsection, agglomerative hierarchical clustering results only a hierarchical structure not the best configuration. In order to determine the number of clusters that yields best, in each step of agglomeration pseudo-F statistic [62] is calculated as an indicator of appropriate number of clusters.

Pseudo-F statistic is intended to capture the tightness of data groups, and in essence a ratio of the mean sum of squares between clusters to the mean sum of squares within clusters. Thus, at the appropriate number of groups the pseudo-F statistic is expected to give a peak. The calculation of pseudo-F statistic for a given dataset and groups can be given in (10).

$$Pseudo - F = \frac{(SST-SSW)/(G-1)}{SSW/(S-G)} \quad (10)$$

Where SST is the total sum of squares, SSW is within-cluster sum of squares, S is the total number of samples and G is the number of groups.

Thus at the end of the agglomerative hierarchical clustering algorithm, the number of clusters giving peak pseudo-F statistic is obtained. Since there is a maximum limit in the model library K, number of clusters which is smaller than $K$and results in a peak in pseudo F-statistic is selected as the number of clusters. However, in order to remove outliers before the model learning with EM, the number of training data encapsulated by each cluster found by hierarchical clustering algorithm is inspected. If the number of training data in a cluster is smaller than the minimum required samples N, these data are not used for learning and passed to the next frame. Number of appropriate Gaussian models found at last is used as an input to the EM algorithm and model learning is done.

After the first frame, mentioned agglomerative hierarchical clustering procedure is applied for learning of new models after the model updating scheme described in $Model\ Updating$ section.

The overall model learning and updating scheme with hierarchical clustering is given in Figure 26 as a flow chart.

In order to evaluate the performance effect of agglomerative hierarchical clustering on the algorithm, the algorithm is run with hierarchical clustering and without hierarchical clustering with five different configurations and the results are given in Table 2.

Table 2 - Classification results of the proposed algorithm for five different sets of algorithm parameters with and without agglomerative hierarchical clustering.

		without Hierarchical Clustering				**with Hierarchical Clustering**			
		TP %	**TN %**	**FP %**	**FN %**	**TP %**	**TN %**	**FP %**	**FN %**
Config 1	**Ave %**	70.98	93.48	6.52	29.02	68.99	93.45	6.55	31.01
	Std Dev	19.97	5.72	5.72	19.97	21.70	6.43	6.43	21.70
Config 2	**Ave %**	86.53	83.92	16.08	13.47	84.84	85.09	14.91	15.16
	Std Dev	16.30	11.74	11.74	16.30	17.58	10.26	10.26	17.58
Config 3	**Ave %**	91.81	75.17	24.83	8.19	91.05	76.40	23.60	8.95
	Std Dev	15.37	15.36	15.36	15.37	15.78	14.84	14.84	15.78
Config 4	**Ave %**	58.64	97.52	2.48	41.36	55.08	97.92	2.08	44.92
	Std Dev	22.02	3.10	3.10	22.02	21.76	2.73	2.73	21.76
Config 5	**Ave %**	67.88	96.14	3.86	32.12	60.92	96.92	3.08	39.08
	Std Dev	21.25	4.12	4.12	21.25	22.71	3.68	3.68	22.71

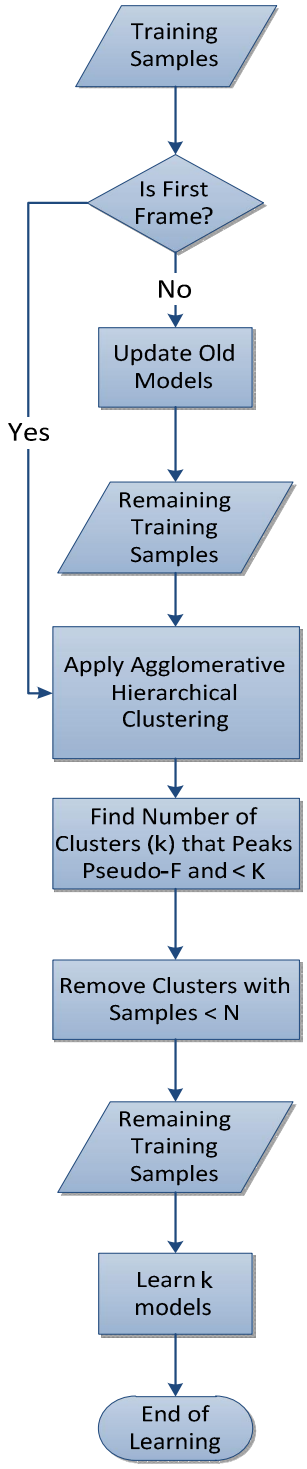

Figure 26 - Flow chart of learning & updating models with agglomerative hierarchical clustering.

According to the results, it can be said that hierarchical clustering decreases false positive rate (FP) very slightly while decreasing true positive rate (TP) much more considerably in general. Although it damages the algorithm success considerably in some configurations, it has a very slight effect when true positive rates are high. Considering its high algorithmic complexity (for c training samples $O(c^2)$ for hierarchical clustering and $O(c)$ for the calculation of Pseudo-F statistic at best), it can be said that there is no need to use agglomerative hierarchical clustering algorithm to determine the number of clusters to be learned in each frame. Another conclusion that can be drawn from these results is that the success of the algorithm does not depend much on the number of clusters to be learned in the first frame (k_0) and in each new frame (k). In another words, k_0 and k have a negligible effect on algorithm performance.

3.4.6 Classification

After the models learned, the last step is the classification of all patches in the image using Gaussian color distribution models in the model library. The classification of patches is done using feature vectors extracted from image patches, that is, the H-S joint histograms of m bins. According to the similarity of a feature vector to any of the models in the model library, the patch related to that feature vector is accepted either as road or rejected. Mathematically, the similarity can be represented as the distance of the feature vector of the patch to any of the model learned in the feature space. After this distance is calculated, according to a classification threshold (T_c), the patch with the corresponding feature vector can be classified as road or non-road. If the distance between the feature vector and the model is smaller than the threshold value T_c it is classified as road, else it is classified as non-road. There are two problems arise in classification. The first one is the distant places that are not connected to road but has a color distribution and smoothness like road regions and the second one is the selection of right similarity (or distance) measure.

In order to solve the first problem, in the classification of each frame, the training samples generated in that frame are used as seeds for classification. After these seeds are initialized, classification is applied by region growing. Thus, to be accepted as road, the patch both satisfies the similarity measure as well as the connectivity to the training samples. In the first step of region growing, four-neighbourhood of training samples are queried for the similarity to the models. After that in each step, region growing continues with the four-neighborhood of the patches classified in the previous step. This loop continues until there are no new patches are classified as road. The procedure can be visualized as in Figure 27.

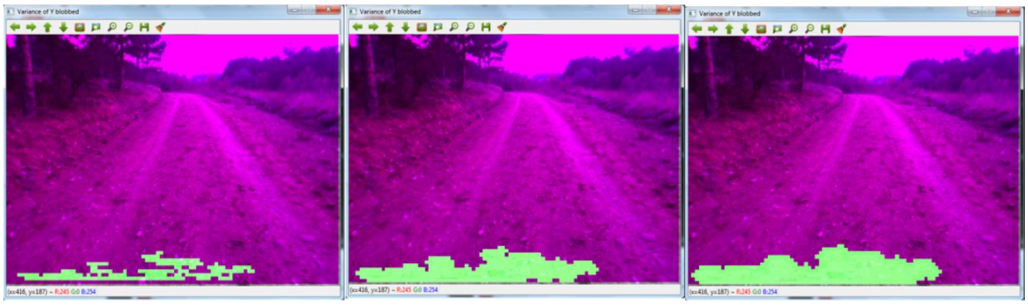

Figure 27 - Steps of classification via region growing. (Left) Training samples obtained in the frame as seeds for classification. (Middle) Result of region growing after the first step. (Right) Result of the region growing after second step.

For the solution of the second problem, the classification performance of the algorithm with six different distance measure is compared. The distance measures used for the classification step are; Manhattan distance, Euclidian distance, Mahalanobis distance, Chebyshev distance, Hellinger distance and Chi-square distance measures.

3.4.6.1 Manhattan Distance

Manhattan distance is a distance metric, in which the distance between two vectors is the sum of the absolute differences of their coordinates in the space. The metric is also known as L_1 norm or city block distance. For two vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ of dimensions d, it can be expressed as given in (11) [63].

$$dist_1(\boldsymbol{p},\boldsymbol{q}) = \sum_{i=1}^{d}|p_i - q_i| \tag{11}$$

Since it is required to find the distance between a feature vector and a model in this algorithm, for the feature vector $\boldsymbol{p}$ and mean $\boldsymbol{\mu}$ of the Gaussian model $\boldsymbol{M}$, (11) can be modified as given in (12).

$$dist_1(\boldsymbol{p},\boldsymbol{M}) = \sum_{i=1}^{d}|p_i - \mu_i| \tag{12}$$

3.4.6.2 Euclidian Distance

Euclidian distance is the ordinary distance metric that can be found by Pythagorean formula. The Euclidian distance between two vectors is actually the length of the line segment connecting them. For two vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ of dimensions d, it can be expressed as given in (13) [63].

$$dist_E(\boldsymbol{p},\boldsymbol{q}) = \sqrt{\sum_{i=1}^{d}(p_i - q_i)^2} \tag{13}$$

Then the distance between the feature vector $\boldsymbol{p}$ and mean $\boldsymbol{\mu}$ of the Gaussian model $\boldsymbol{M}$, (13) can be expressed as given in (14).

$$dist_E(\boldsymbol{p}, \boldsymbol{M}) = \sqrt{\sum_{i=1}^{d}(p_i - \mu_i)^2} \tag{14}$$

3.4.6.3 Mahalanobis Distance

Mahalanobis distance metric is a distance measure based on the correlations between variables [63]. Since it takes into account the correlations of the dataset and is scale-invariant, it is different from Euclidian distance measure. For a dataset D with mean $\boldsymbol{\mu_D}$ and covariance $\boldsymbol{S_D}$ of dimension d, the calculation of Mahalanobis distance for a vector $\boldsymbol{p}$ is given in (15).

$$dist_M(\boldsymbol{p}) = \sqrt{(\boldsymbol{p} - \boldsymbol{\mu_D})^T \boldsymbol{S_D^{-1}} (\boldsymbol{p} - \boldsymbol{\mu_D})} \tag{15}$$

If covariance matrix $\boldsymbol{S_D}$ is a diagonal, then the resulting distance measure is called normalized Euclidian distance and if covariance matrix $\boldsymbol{S_D}$ is an identity matrix, then the distance measure is reduced to Euclidian distance. For a feature vector $\boldsymbol{p}$ and mean $\boldsymbol{\mu}$, covariance $\boldsymbol{\Sigma}$ of the Gaussian model $\boldsymbol{M}$, (15) can be expressed as given in (16).

$$dist_M(\boldsymbol{p}, \boldsymbol{M}) = \sqrt{(\boldsymbol{p} - \boldsymbol{\mu})^T \boldsymbol{\Sigma^{-1}} (\boldsymbol{p} - \boldsymbol{\mu})} \tag{16}$$

3.4.6.4 Chebyshev Distance

Chebyshev distance metric, also known as maximum metric or L_∞ metric [64], is a metric defined as the distance between two vectors is the greatest of their differences along any dimension of the feature space where they defined. Mathematically, for two vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ of dimensions d it can be expressed as given in (17).

$$dist_\infty(\boldsymbol{p}, \boldsymbol{q}) = \max_i(|p_i - q_i|) \tag{17}$$

Then the distance between the feature vector $\boldsymbol{p}$ and mean $\boldsymbol{\mu}$ of the Gaussian model $\boldsymbol{M}$, (17) can be expressed as given in (18).

$$dist_\infty(\boldsymbol{p}, \boldsymbol{M}) = \max_i(|p_i - \mu_i|) \tag{18}$$

3.4.6.5 Hellinger Distance

Hellinger distance is a distance measure to quantify the similarity between two distributions. Since in the proposed algorithm, the feature vectors and Gaussian model means are actually H-S joint histograms, they are actually distributions. Thus, Hellinger distance metric can be used to calculate the similarity (or dissimilarity) between them. For the feature vector $\boldsymbol{p}$ and the mean $\boldsymbol{\mu}$

of the Gaussian model $\boldsymbol{M}$ of dimensions d, the calculation of Hellinger distance between the feature vector and the model can be found as given in (19) [65].

$$dist_H(\boldsymbol{p}, \boldsymbol{M}) = \sqrt{1 - \frac{1}{\sqrt{\bar{\boldsymbol{p}}\bar{\boldsymbol{\mu}}d^2}} \sum_{i=1}^{d} \sqrt{p_i \mu_i}} \tag{19}$$

Where $\bar{\boldsymbol{p}}$ and $\bar{\boldsymbol{\mu}}$ are given in (20) and (21), respectively.

$$\bar{\boldsymbol{p}} = \frac{1}{d} \sum_{i=1}^{d} p_i \tag{20}$$

$$\bar{\boldsymbol{\mu}} = \frac{1}{d} \sum_{i=1}^{d} \mu_i \tag{21}$$

3.4.6.6 Chi-squared Distance

Another distance measure to quantify the similarity between two distributions is chi-sqaured distance measure. Chi-squared distance measure is used extensively in computer vision to measure the dissimilarity or similarity between two histograms. . For the feature vector $\boldsymbol{p}$ and the mean $\boldsymbol{\mu}$ of the Gaussian model $\boldsymbol{M}$ of dimensions d, the calculation of Chi-squared distance between the feature vector and the model can be found as given in (22) [65].

$$dist_C(\boldsymbol{p}, \boldsymbol{M}) = \sum_{i=1}^{d} \frac{(p_i - \mu_i)^2}{p_i} \tag{22}$$

3.4.6.7 Results

The classification performance of mentioned six distance measures for the same configuration with increasing classification threshold (T_c) is given in Figure 28. In Figure 28, the average false positive (FP) ratio versus the average true positive (TP) ratio of the algorithm is given. Since true positives are desirable and false positives are undesirable, the FP vs. TP curve should be as steep as possible. As the results suggests, the steepest curve is belonged to Mahalanobis distance (applied as Normalized Euclidian Distance here since covariance matrix is diagonal). Thus, Mahalanobis distance metric is selected in the use of classification due to its maximized performance. The worst case time complexity of classification all patches with Mahalanobis distance measure is $O(PKm^6)$. Since the covariance matrix is assumed to be diagonal, the Mahalanobis distance measure becomes Normalized Euclidian distance with a worst case time complexity $O(PKm^4)$.

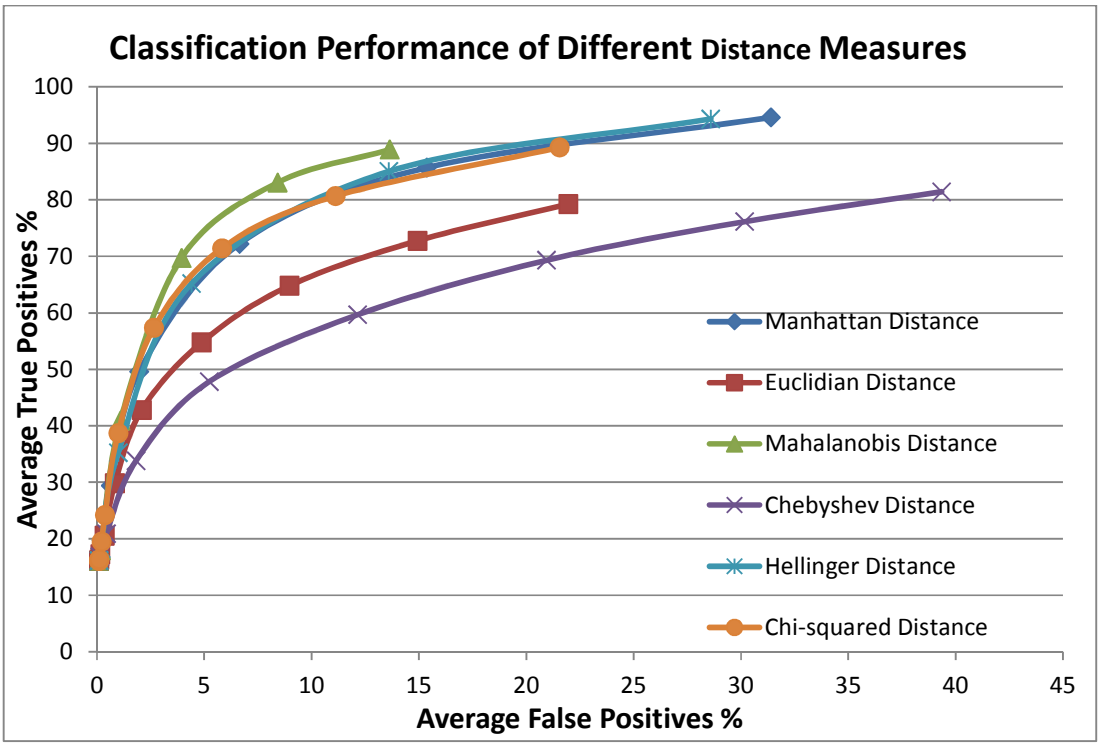

Figure 28 - Classification performance of different distance measures for the same configuration with increasing classification threshold (T_c).

3.5 Experiments and Results

In this section the performance of the algorithm is evaluated. For a configuration selected as reference, each of the algorithm's parameters is changed one at a time to see the effect of the parameters on the performance and the parameters that yield the best performance are tried to find. The reference algorithm parameters selected are given in Table 3.

Table 3 - Reference values for algorithm parameters to evaluate the performance effect of each parameter.

Parameter	$T_r\ (m^2)$	$n\ (pixels)$	$m\ (bins)$	k_0	k	K	N	T_m	T_c
Value	0.00001	5	8	1	1	10	20	1	5

For the evaluation of the algorithm, the data gathered from the pathways of Yalincak village at METU campus – a total of 360 frames- as explained in *Data Gathering* section are used. The data gathered are labeled as road and non-road manually by using a custom interface developed in Visual C# [52] and with AForge.NET image processing libraries [54]. A screenshot taken from the software with manually labeled road region is presented in Figure 29. The white regions in the image in Figure 29 are labeled as road while other regions are labeled as non-road. The labeled images are used as ground truth to compare the images labeled by the algorithm.

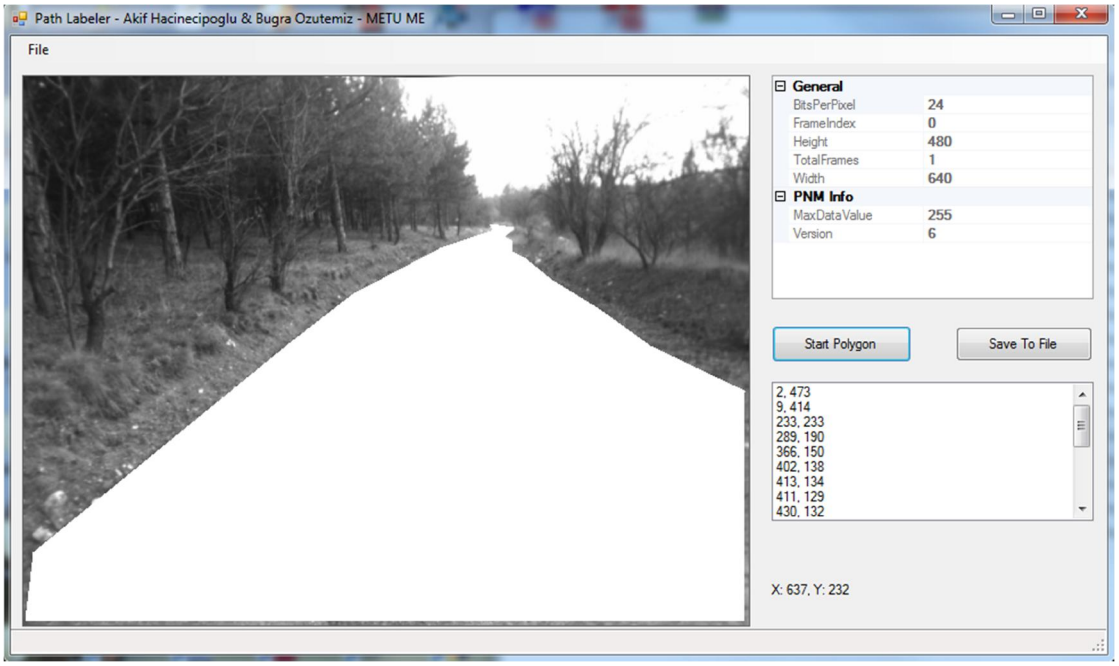

Figure 29 - Road region labeling user interface for ground truth data generation.

As performance metrics, average (mean) and standard deviation of true positive (TP) and false positive (FP) rates of all 360 frames are used. The average TP and FP rates of all 360 frames give the overall performance of the algorithm. TP rate shows the rate of detection of road regions successfully. FP rate, on the other hand, shows the rate of detection of non-road regions successfully. Thus, it is desired to have the highest possible average TP rate while having the smallest possible average FP rate to have safe road detection. If FP rate is too high, this means the non-road regions are classified as road regions which is highly dangerous. While looking at the average FP-TP plots, the parameters resulting highest TP/FP ratio should be selected to obtain the highest performance. The standard deviations of TP and FP rates of all 360 frames, on

the other hand, show the variation of frame-based TP and FP rates. Hence, both the standard deviation of TP and FP rates should be as small as possible for reliability and faster learning when the road appearance changes. Through the experiments and parameter selection, the highest allowable FP rate is selected as 10% for a satisfactory performance. Thus, the parameters are selected to obtain the summation of average FP and standard deviation of FP as 10% at maximum.

The first experimentation is done to see the effect of the patch size on the algorithm's road detection performance. The change of average FP rate vs. average TP rate with the change in patch size from 2 to 9 pixels is given in Figure 30. The results say that as the patch size increases, average FP rates are decreasing as well as average TP rates. It is expected due to the fact that increasing patch size increases the distinctivity of the algorithm. It also increases the sparsity within the feature space. On the other hand, the increase in the patch size also increases the subsampling and number of training samples obtained in each frame, and hence decreases the average TP rates.

The increase in the patch size also increases the height variance within a patch and as a result for the same height variance threshold lesser training samples are obtained. In fact, after some point the increase in the patch size only decreases average TP rates without changing average FP rates. This situation can be seen in Figure 30 after a patch size of 7 pixels.

In Figure 31, the standard deviations of FP and TP rates are given. As figure suggests, the standard deviation of TP rates grows steadily as patch size increases while the standard deviation of FP rates decreases up to a patch size of 7 pixels, but increases again after 7 pixels. As it has been mentioned this characteristics is a direct result of obtaining a very small number of training samples after a patch size of 7 pixels.

As a result of the first experimentation it can be said that patch size (n) has a major effect on algorithm performance and a patch size of 5x5 or 6x6 seems like a reasonable compromise between FP and TP rates.

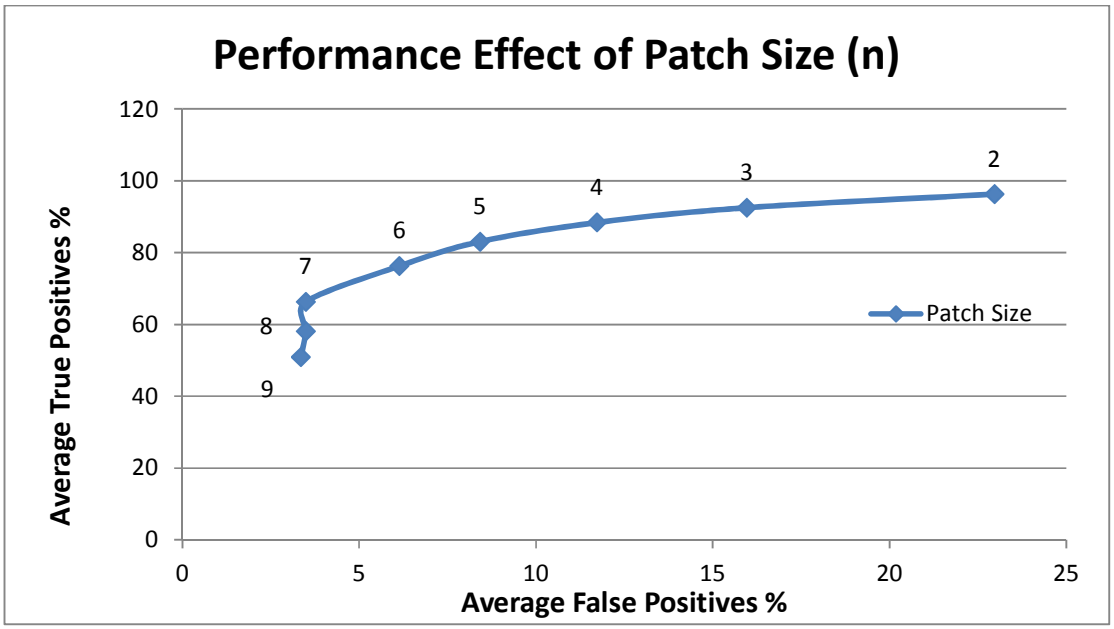

Figure 30 - The effect of patch size on average TP rates and average FP rates of the algorithm. Note that the number of pixels consisted by a patch is nxn.

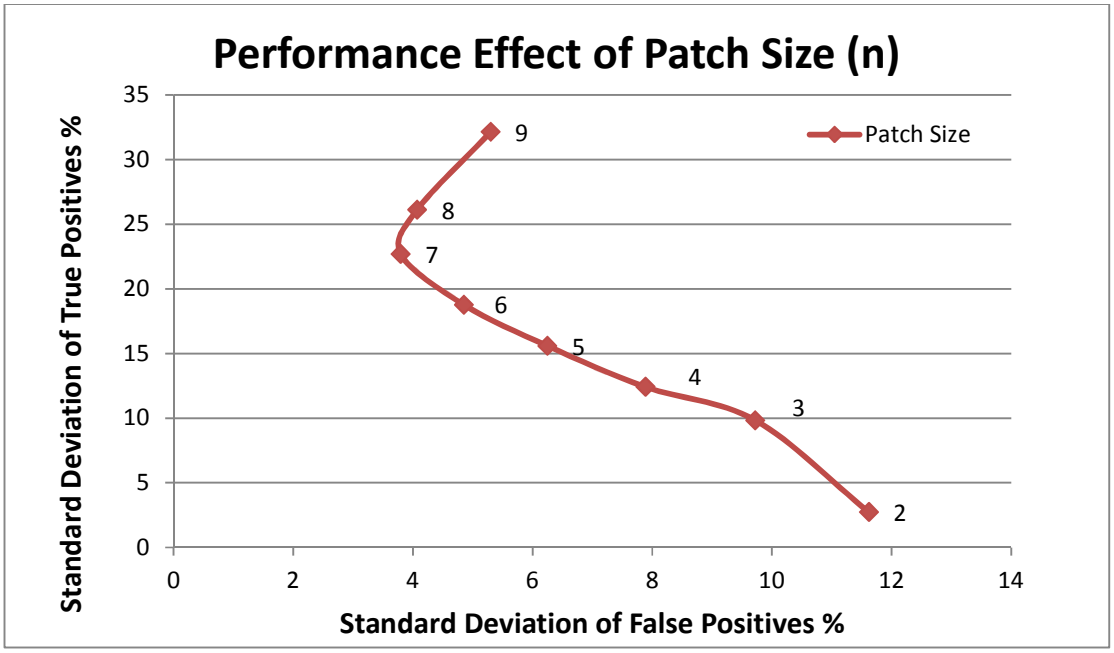

Figure 31 - The effect of patch size on the standard deviations of TP and FP rates of the algorithm. Note that the number of pixels consisted by a patch is nxn.

The second experiment is done to see the effect of the classification threshold (T_c) on the algorithm performance. The change of average FP rate vs. average TP rate with the change in classification threshold from 0.5 to 6 is given in Figure 32. Since classification threshold (T_c) directly affects which patch is accepted or rejected, as the threshold increases both average FP rates and TP rates increase. However as the figure suggests, after some point the increase in the average TP rate becomes insignificant compared to the increase in the average FP rate.

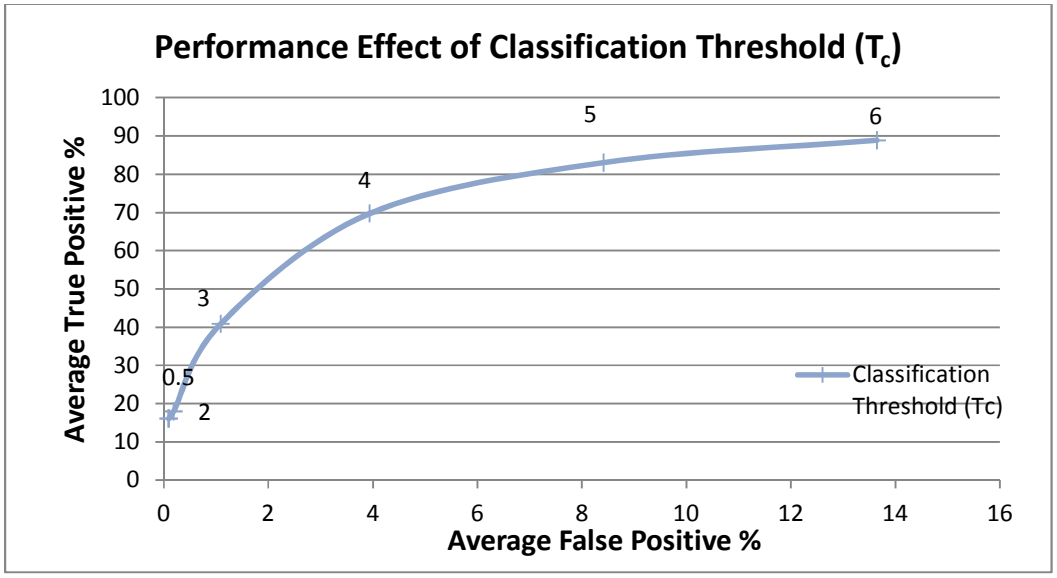

Figure 32 - The effect of classification threshold on average TP rates and average FP rates of the algorithm.

In Figure 33, the standard deviation of FP rate vs. TP rate with the change in classification threshold (T_c) is given. As it can be seen from the figure the uncertainty in the TP rate first increase and then decrease while the uncertainty in the FP rate continuously grows. Thus, to hold the uncertainty small the figure suggests choosing a relatively small classification threshold value.

Since selection of a satisfactory classification threshold (T_c) also depends on the feature space. It is more suitable to select the classification threshold with other parameters.

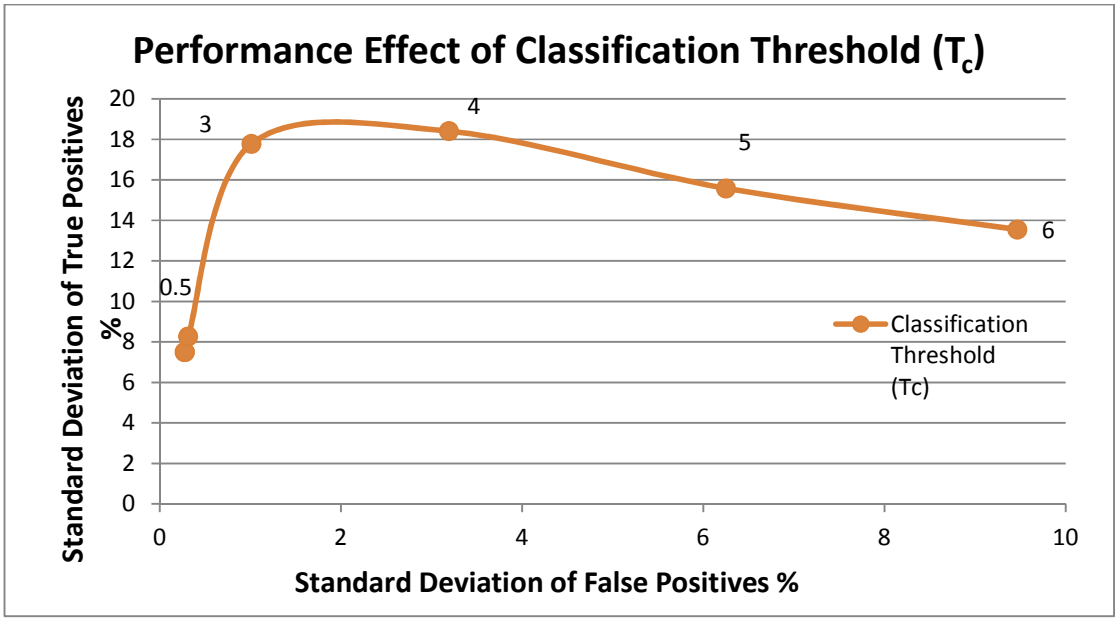

Figure 33 - The effect of classification threshold on the standard deviations of TP and FP rates of the algorithm.

After the classification threshold (T_c) the next experimentation is done to investigate the effects of model update threshold (T_m) on the algorithm classification performance. This parameter directly affects the shape of the Gaussians. As the threshold increases, the learned Gaussians become wider and as the threshold decreases, the models become more compact and hence the uncertainty within the models decreases.

In Figure 34, the average of FP rate vs. TP rate with the change in model update threshold (T_m) is given. As the figure suggests, as the update threshold increases both average TP and FP rates are increasing. However, after some point the increase in the average FP rate is faster. On the other hand, the change in standard deviations of TP and FP rates with respect to the change in update threshold is somewhat different. As the update threshold increases, the change in the standard deviation of TP rate is very small compared to the large change in the standard deviation of FP rate. This figure suggests the selection of a small update threshold value.

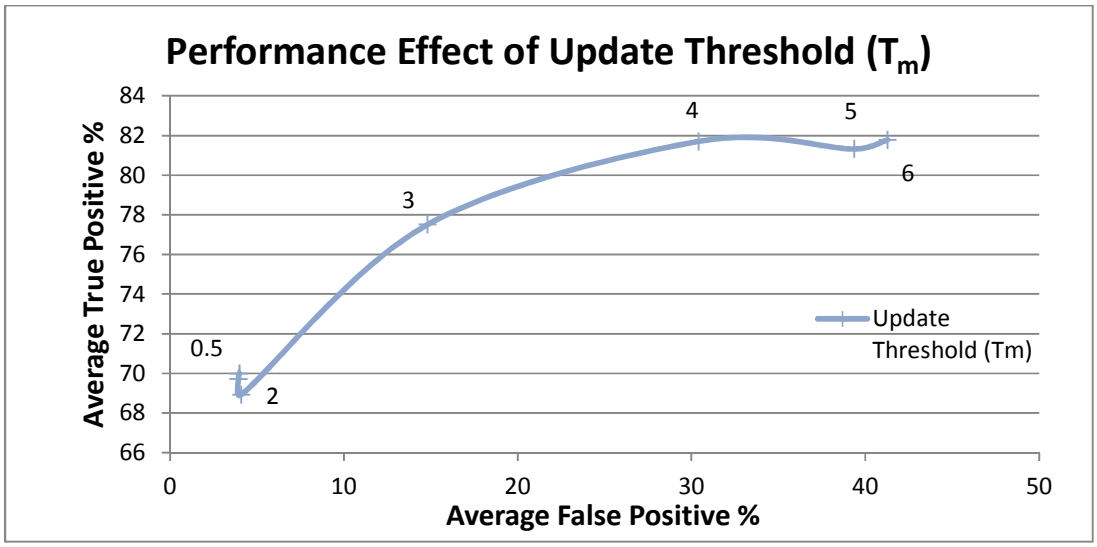

Figure 34 - The effect of the update threshold on the average TP and FP rates of the algorithm.

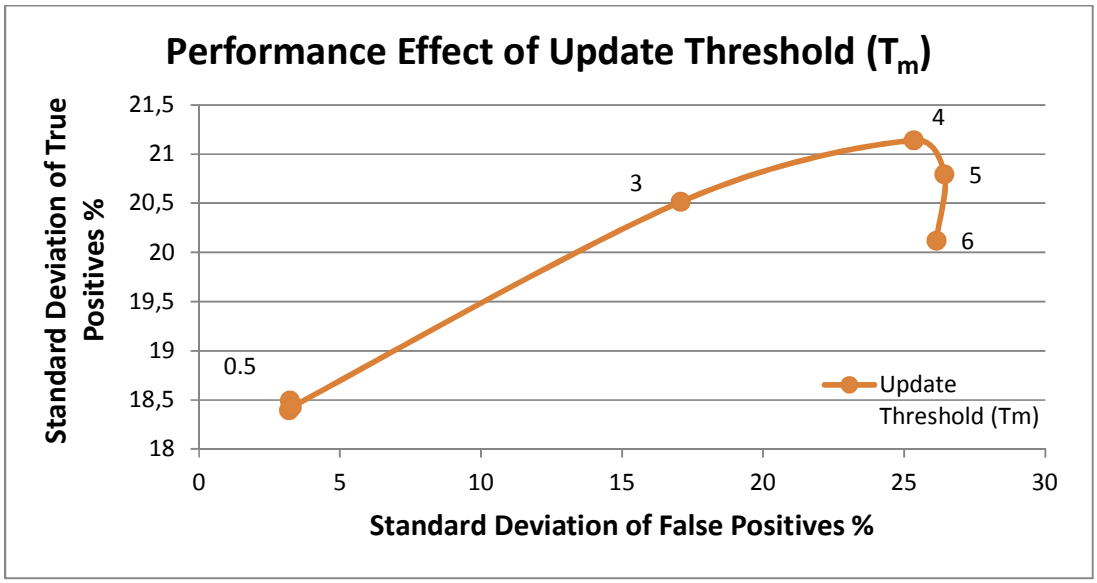

Figure 35 - The effect of the update threshold on the standard deviations of TP and FP rates of the algorithm.

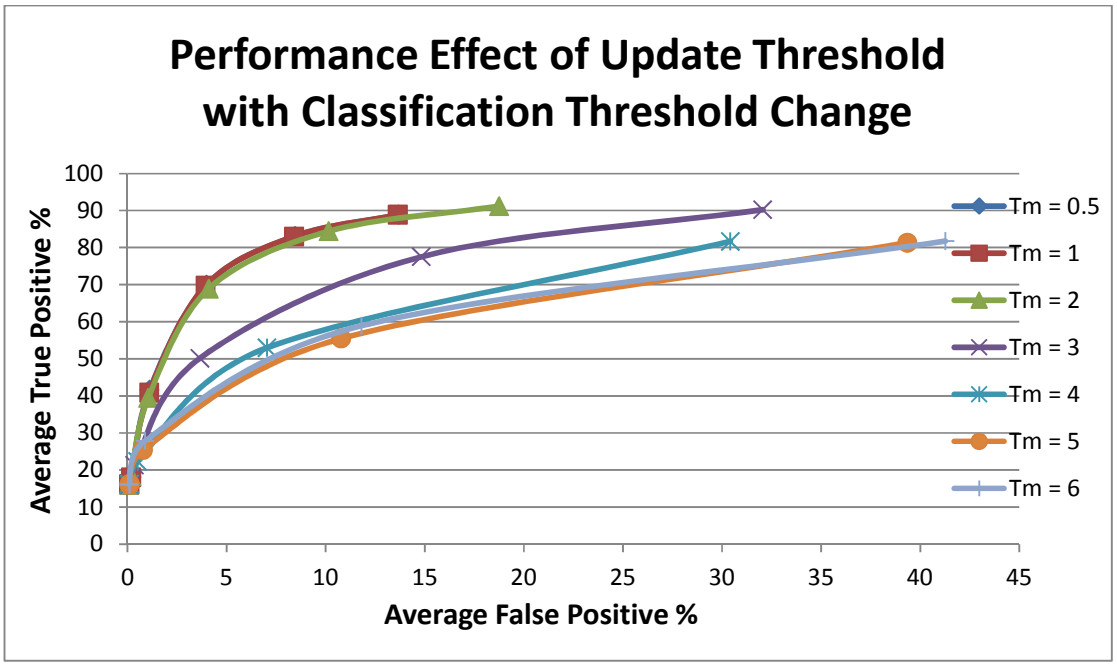

Figure 36 - The effect of model update threshold with classification threshold change on average TP rates and average FP rates of the algorithm.

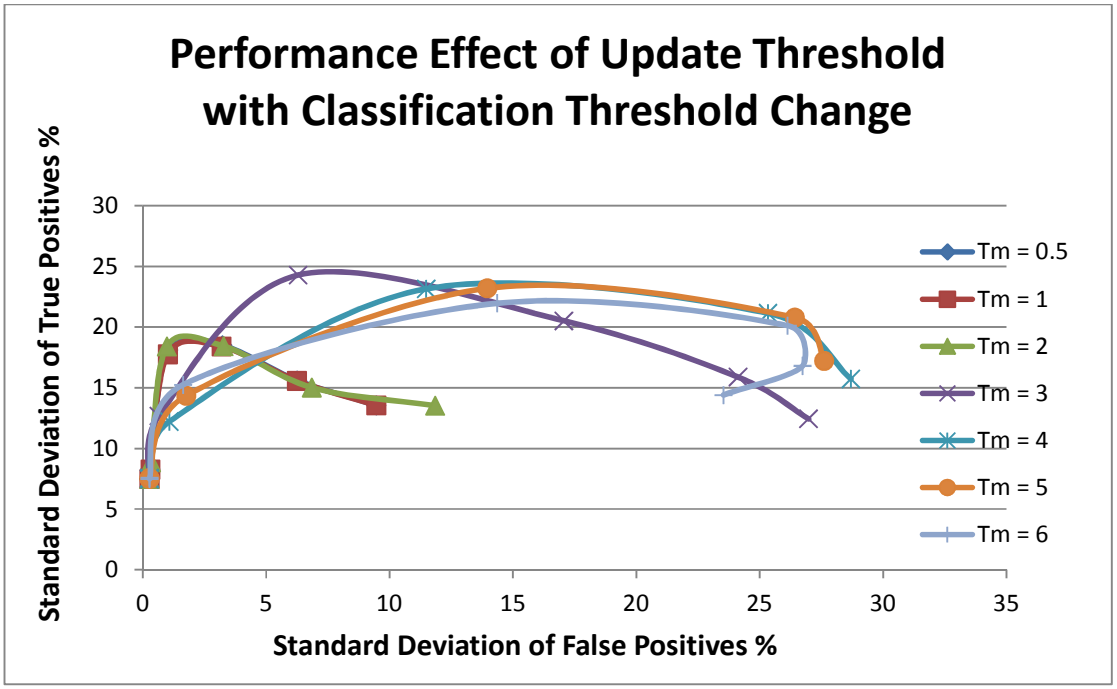

Figure 37 - The effect of model update threshold and classification threshold change on the standard deviations of TP and FP rates of the algorithm.

If besides changing model update threshold (T_m), classification threshold (T_c) is also changed, the behavior of the system becomes as it is given in Figure 36 and Figure 37. As figures suggest, as classification threshold (T_c) for a given model update threshold (T_m) both average TP and FP rates increase, but for small (T_m) the increase in the average FP rate with respect to the average TP rate is small. For the standard deviations of TP and FP, as (T_c) increases the standard deviation of TP first increases but then decreases while the standard deviation of FP always increases. Again, for small (T_m) the increase in the standard deviation of FP with respect to TP is small. According to the figures, it seems logical to select (T_m) something smaller than 2.

The next experimentation is done on the performance effects of the number of histogram bins (m). Since the feature vector used for the classification and learning is the H-S joint histogram of each image patch the number of bins (m) in one channel is actually determines the size of the feature space. Thus, if the histogram of one channel has (m) bins, then the joint histogram has (m^2) bins and the feature space becomes (m^2) dimensional.

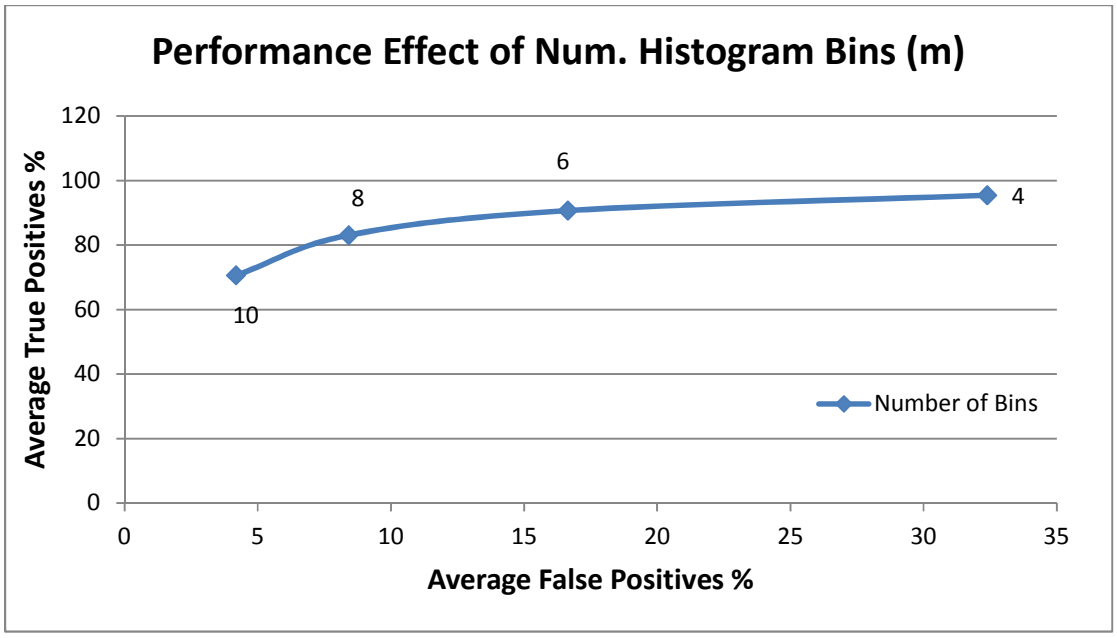

Figure 38 - The effect of number of histogram bins on average TP rates and average FP rates of the algorithm.

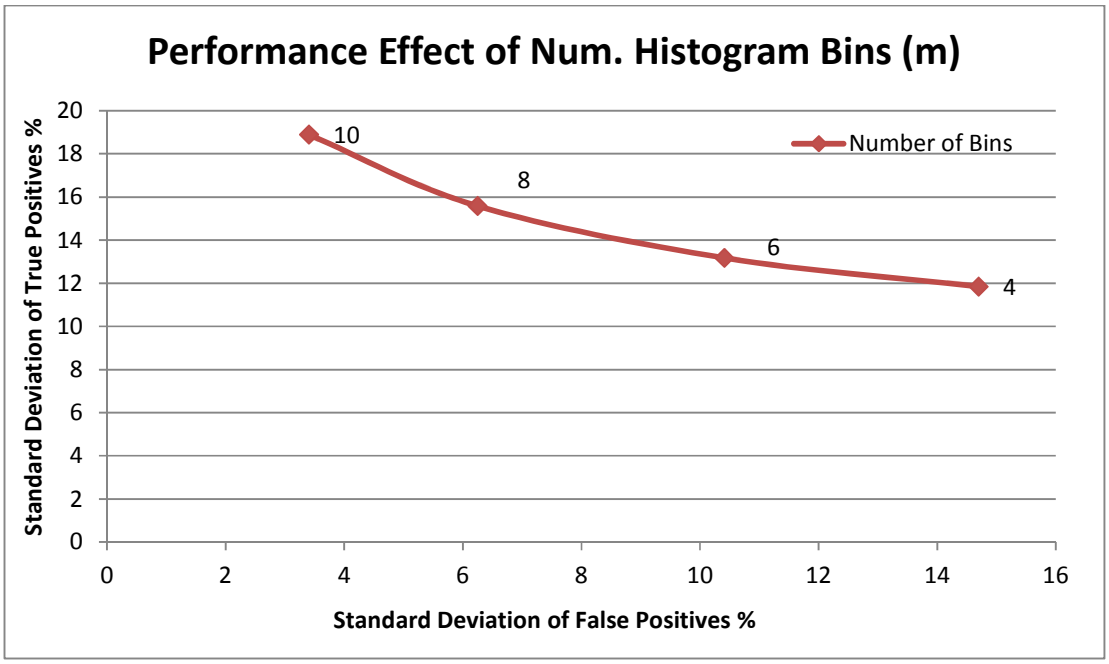

Figure 39 - The effect of number of histogram bins on the standard deviations of TP and FP rates of the algorithm.

If only the number of histogram bins (m) is changed while other parameters are held constant, the FP vs. TP rates of the algorithm has a trend as given in Figure 38 and Figure 39. According to the figures, it can be said that the average FP and TP rates of the algorithm decrease as number of bins increases. It is expected due to the increase in the sparsity as a result of high dimensional feature space. In standard deviations however, as the number of bins increases the uncertainty in the false positives decreases but the uncertainty in the true positives increases. Looking only these figures suggest the selection of 8 or 10 bins might yield a reasonable performance.

If besides changing number of bins (m), classification threshold (T_c) is also changed the figures becomes surprisingly different and given in Figure 40 and Figure 41. As Figure 40 suggests, with a proper selection of classification threshold (T_c) approximately same average FP and TP rate performances can be obtained within a different number of histogram bins (m). As for the standard deviation of FP and TP rate performances given in Figure 41, the characteristics are similar but the standard deviation of TP is smaller in smaller number of bins. According to these results, number of bins (m) can be selected as 6 or maybe 8 to decrease the uncertainty sufficiently.

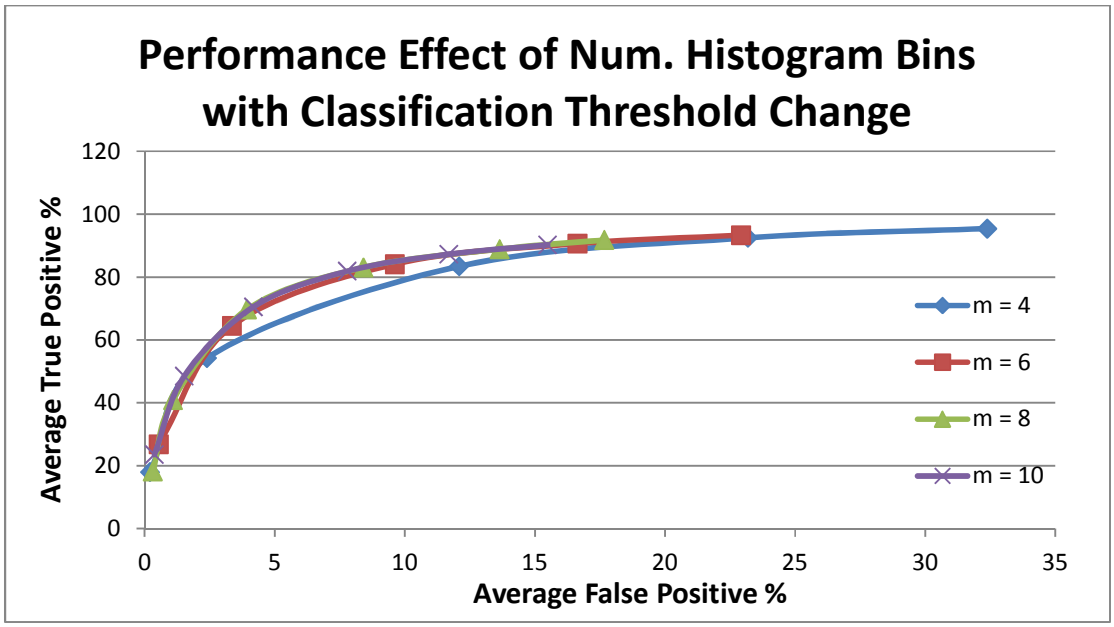

Figure 40 - The effect of number of histogram bins with classification threshold change on average TP rates and average FP rates of the algorithm.

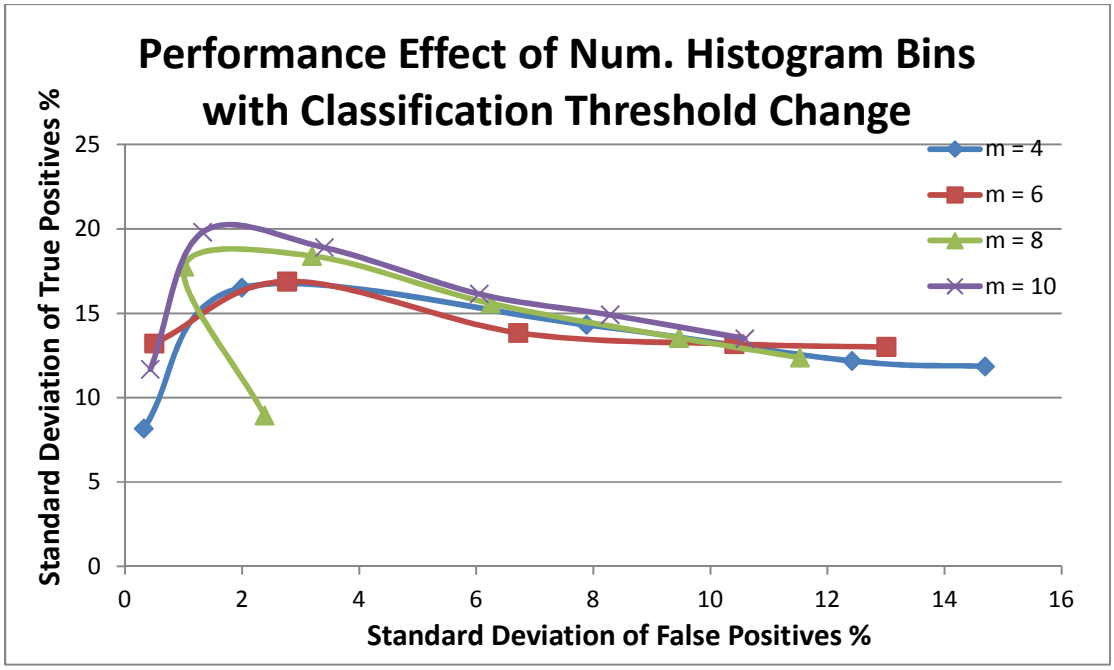

Figure 41 - The effect of number of histogram bins and classification threshold change on the standard deviations of TP and FP rates of the algorithm.

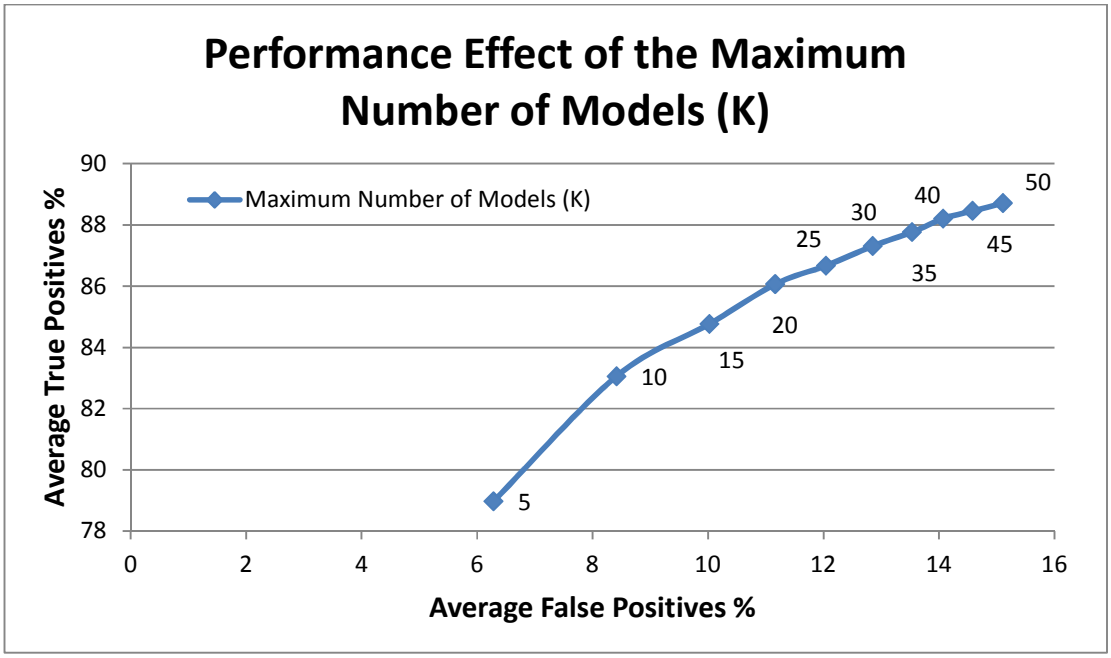

Figure 42 - The effect of maximum number of Gaussian models on average TP rates and average FP rates of the algorithm.

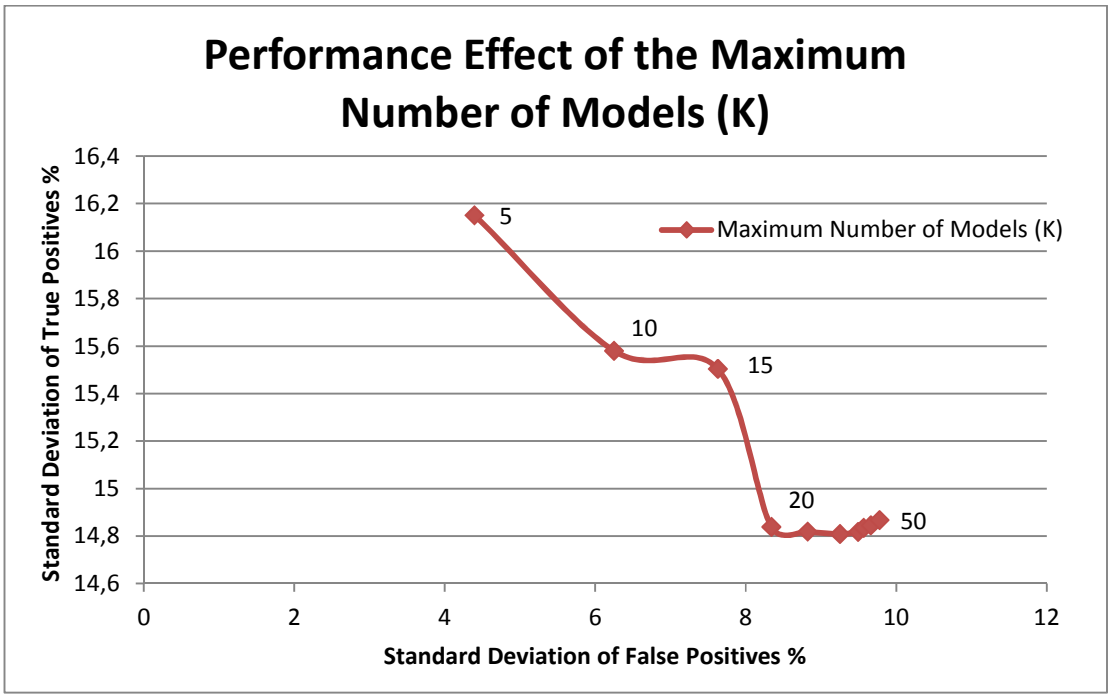

Figure 43 - The effect of maximum number of Gaussian models on the standard deviations of TP and FP rates of the algorithm.

The next experiment is done to see the effect of the maximum size of the Gaussian model library. This parameter determines how much Gaussian color models to be stored and used to classify the given frame. Thus it represents the size of the memory holding roads passed.

The average and standard deviation of FP vs. TP rates of the algorithm are given in Figure 42 and Figure 43. As the average FP vs. TP rates figure suggests, the increase in the maximum number of models (K) results in an increase in both average FP and TP rates. However, the increase in the average FP rates is much more serious compared to the increase in the average TP rates. When the standard deviation of FP vs. TP rates considered, as number of models (K) increase standard deviation of TP rates decreases very slightly while standard deviation of FP rates increases significantly. As a result selecting (K) as 5 or 10 seems logical according to the figures.

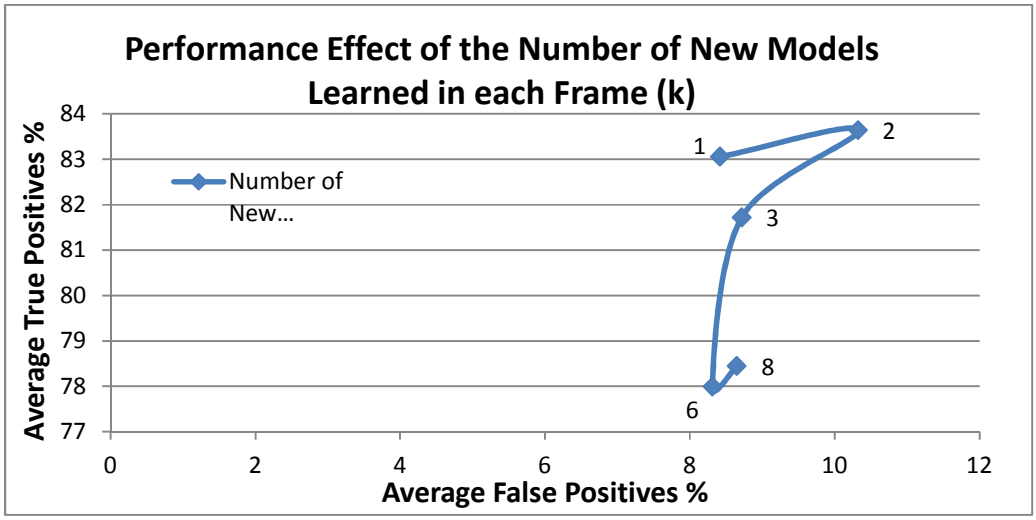

Figure 44 - The effect of number of newly learned Gaussian models in each frame on average TP rates and average FP rates of the algorithm.

The next experiment is done on the effect of the number of newly learned Gaussian models in each frame (k). The performance effects on average FP vs. TP rates and standard deviation of FP vs. TP rates are given in Figure 44 and Figure 45. As the results drawn in the figures suggest, the effect of the parameter is lesser than the effect of (K) and does not change the algorithm performance much. As the value of the parameter grows, that is, the number of color models learned new increases, the algorithm forgets the old roads faster and hence, the average TP rate decreases without much change in average FP rate. In addition to that, as the number of models learned with the same number of training samples increases, the uncertainty per model grows.

Hence, this situation yields an increase in the standard deviation of TP rate without much change in the standard deviation of FP rate. Hence, selecting (k) as 1 seems like an appropriate choice in general.

Another experiment is done to see the effect of the number of initially learned Gaussian color models on the algorithm classification performance. Since this parameter affects the first couple of frames at the beginning, the effect on the overall performance is insignificant as given in Figure 46 and Figure 47. Thus, it is not logical to evaluate the effect of the parameter on the overall performance but on the startup or transient state.

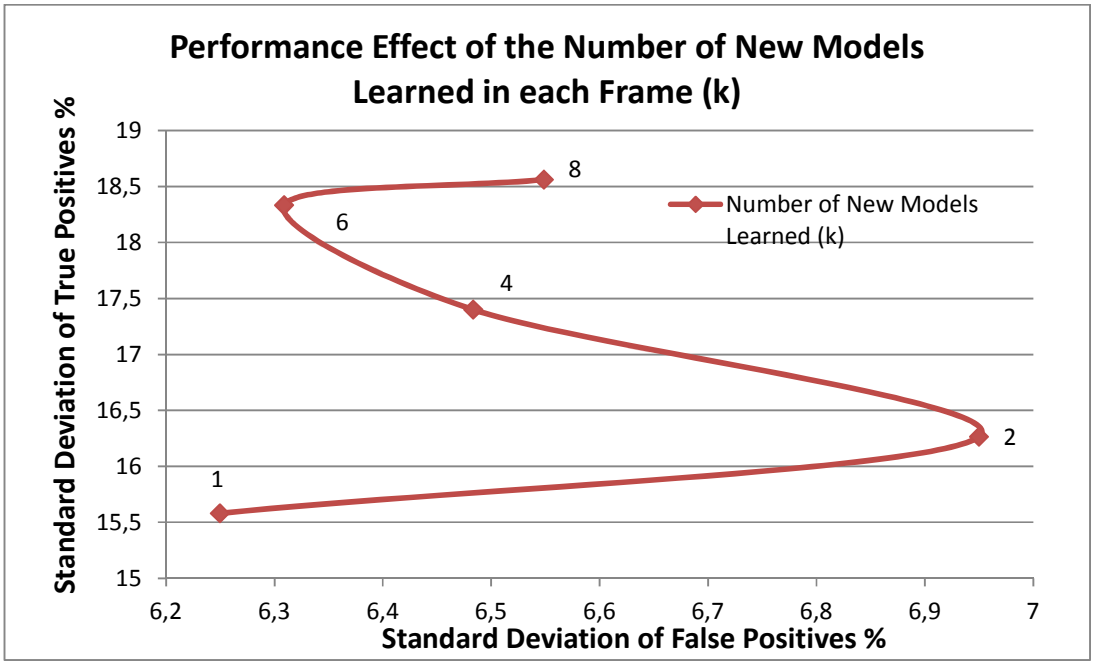

Figure 45 - The effect of number of newly learned Gaussian models in each frame on the standard deviations of TP rates and FP rates of the algorithm.

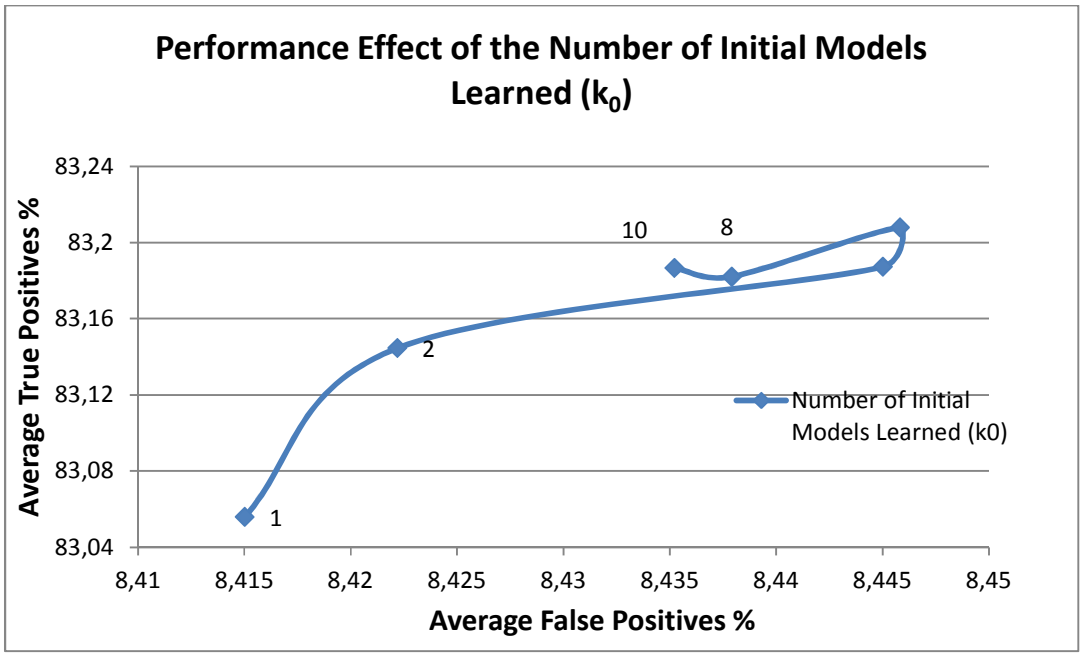

Figure 46 - The effect of number of initially learned Gaussian models on average TP rates and average FP rates of the algorithm.

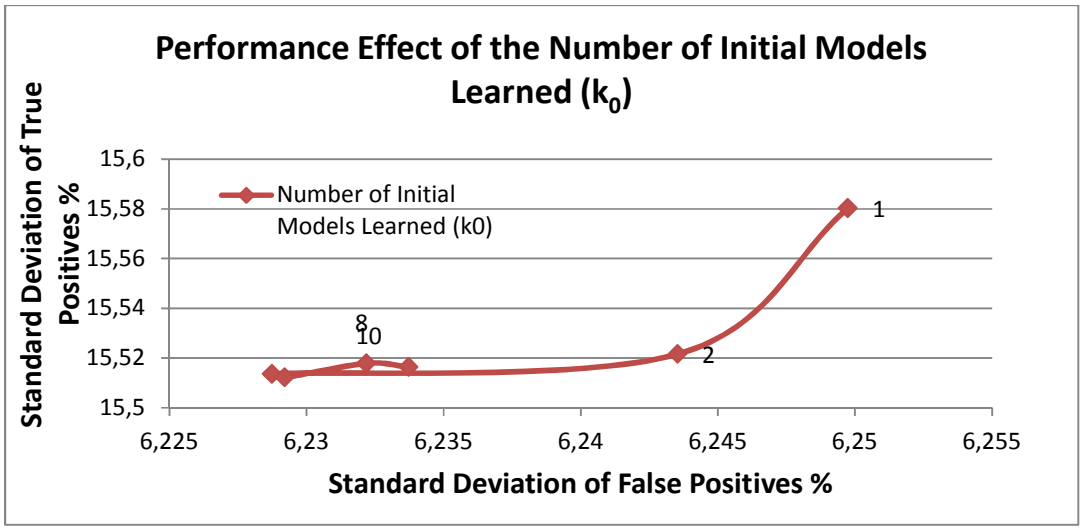

Figure 47 - The effect of number of initially learned Gaussian models on the standard deviations of TP rates and FP rates of the algorithm.

In this respect the performance effect of this parameter on the average and standard deviations of FP and TP rates for the first 10 frames are evaluated as well as the FP and TP rates of the first frame only. As Figure 48 suggests, as (k_0) increases average FP and TP rates increase generally for the first 10 frames. According to Figure 49, as (k_0) increases the uncertainty in TP rate decreases slightly without a significant change in the uncertainty in FP rate. For the first frame, as it can be seen from Figure 50, the trend in Figure 48 is more distinct with a faster increase in the FP rate. According to the trends represented by the figures, it seems logical to select k_0 as 2 for a satisfactory cold start performance of the algorithm.

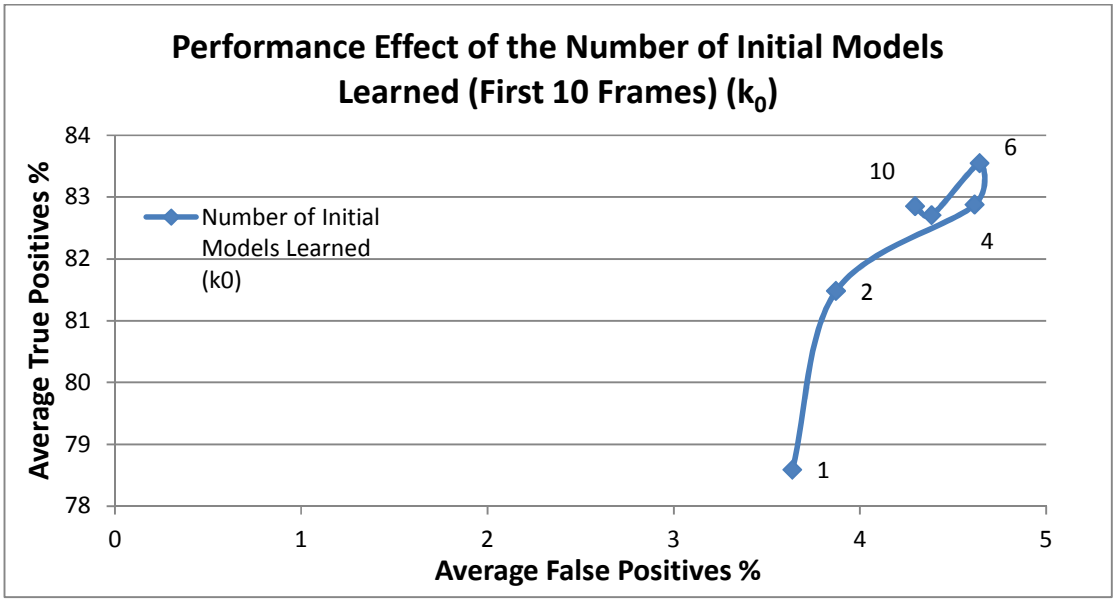

Figure 48 - The effect of number of initially learned Gaussian models on average TP rates and average FP rates of the algorithm for the first 10 frames.

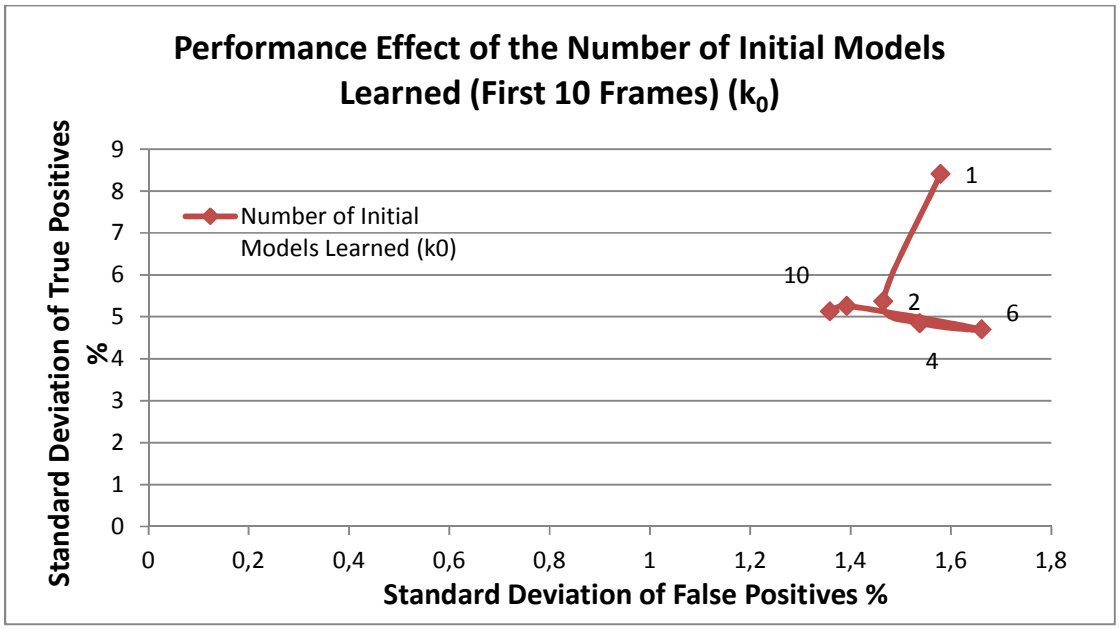

Figure 49 - The effect of number of initially learned Gaussian models on the standard deviations of TP rates and FP rates of the algorithm for the first 10 frames.

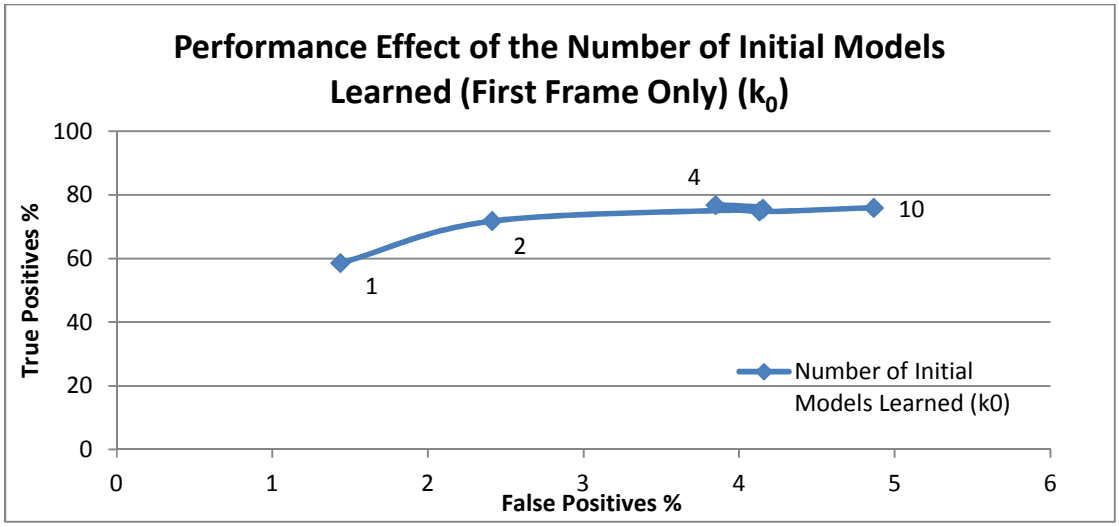

Figure 50 - The effect of number of initially learned Gaussian models on TP and FP rate of the algorithm for the first frame only.

In the next experiment, the effect of the minimum number of training samples (N) required to learn new k Gaussian models is investigated. Again the average TP and FP rates and standard deviation of TP and FP rates are used for the evaluation. The results of the evaluation are given

in Table 4. As the obtained values suggest, the effect of (N) on algorithm performance is negligible and the use of such a minimum value can be omitted.

Table 4 - The effect of the minimum number of training samples required to learn new k models on algorithm performance.

N	Ave. FP %	Ave. TP %	Std. Dev. FP %	Std. Dev. TP %
5	8.41	82.91	6.26	15.60
10	8.43	82.94	6.24	15.60
15	8.43	82.99	6.24	15.58
20	8.42	83.06	6.25	15.58
25	8.43	83.09	6.25	15.57
30	8.43	83.08	6.26	15.64
35	8.43	83.08	6.26	15.64
40	8.43	83.08	6.26	15.64
45	8.45	83.08	6.28	15.66
50	8.45	83.07	6.26	15.66
55	8.45	83.07	6.26	15.66
60	8.45	83.07	6.26	15.66
65	8.45	82.95	6.28	16.03
70	8.45	82.95	6.28	16.03

The last experiment is done to see the effect of roughness threshold (T_r) on the algorithm performance. As the roughness threshold increases, more and more rough surfaces are accepted as training samples and hence the models are learned according to these more rough surfaces. However, selecting this threshold too low will yield an insufficient number of training samples generated and as a result a less general appearance or color model will be generated.

In this study the height is approximated as the camera's vertical axis coordinate. This assumption holds if the camera is held approximately parallel to the ground. Thus, the roughness threshold should also be selected according to the angle between the camera and the ground plane if the camera is not held parallel to the ground.

The average FP vs. TP performance curve of the algorithm is represented in Figure 51. As the threshold value increases, both average TP and FP increase. However, the increase in average TP becomes insignificant after about $T_r = 3e-5$. Thus this figure suggests the selection of the parameter as 2e-5 or 3e-5.

The performance curve related to the standard deviation of FP vs. TP is given in Figure 52. According to the figure, as the roughness threshold increases the standard deviation of TP decreases with a slight increase in the standard deviation of FP. Again, after about $T_r = 4e-5$ the decrease in the standard deviation of TP becomes insignificant. Considering the results given in Figure 51, the selection of the parameter as 2e-5 or 3e-5 seems appropriate.

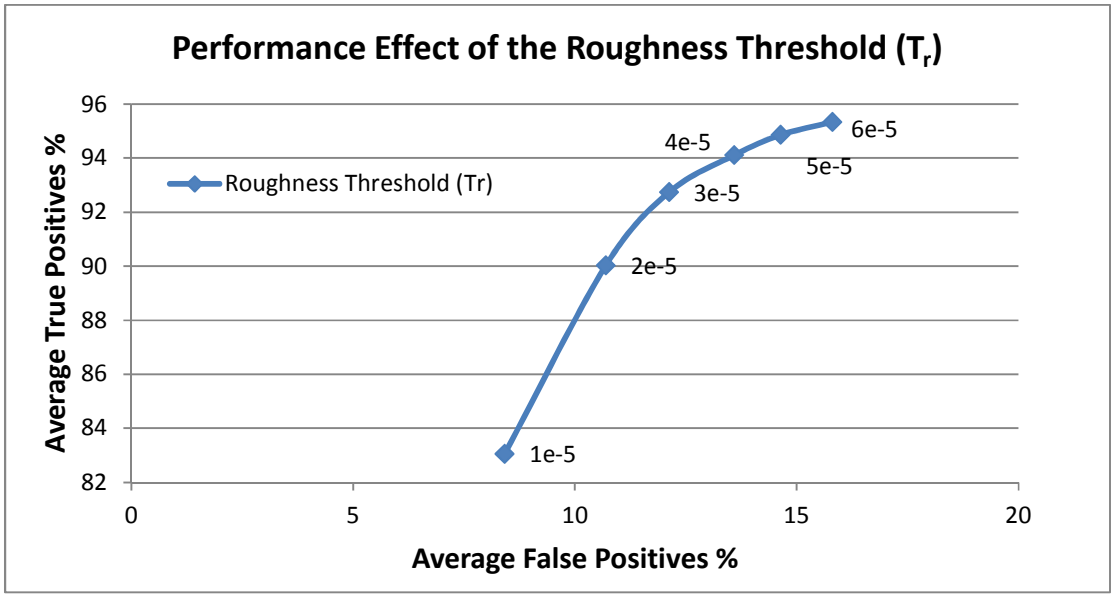

Figure 51 - The effect of roughness threshold on average TP rates and average FP rates of the algorithm.

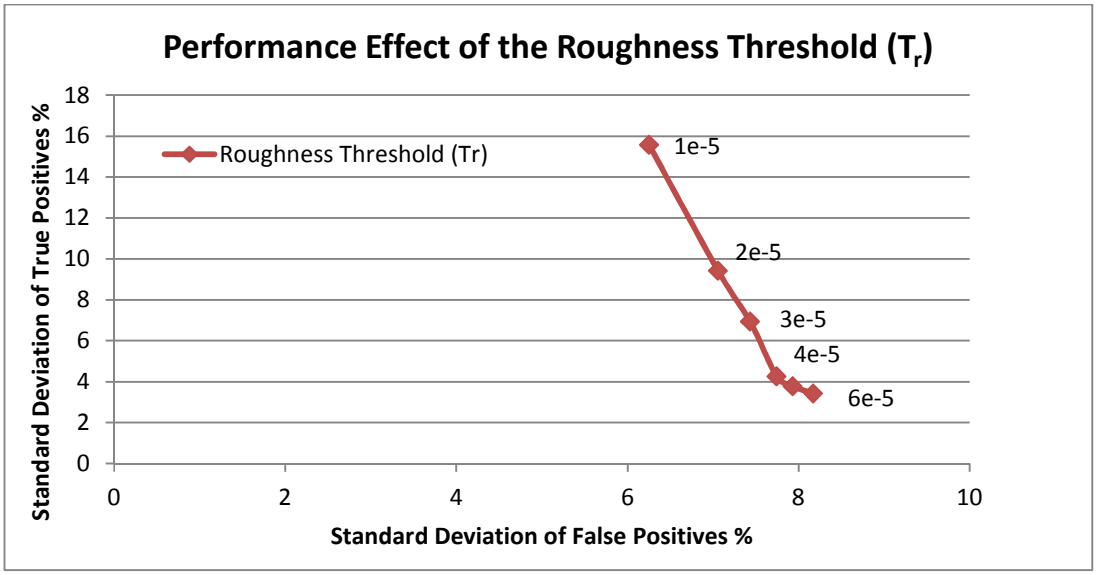

Figure 52 - The effect of roughness threshold on standard deviation of TP rates and standard deviation of FP rates of the algorithm.

Using the results obtained from the analysis, the algorithm parameters are selected as given in Table 5.

Table 5 - Selected algorithm parameters.

Parameter	$T_r\ (m^2)$	$n\ (pixels)$	$m\ (bins)$	k_0	k	K	N	T_m	T_c
Value	0.00003	5	8	2	1	5	20	1	4

Using the parameters given in Table 5 the algorithm is run and the performance of the algorithm is given in Table 6 as average TP, average FP, standard deviation of TP and standard deviation of FP.

Table 6 - Performance results of the selected algorithm parameters over 360 frames.

Ave. FP %	Ave. TP %	Std. Dev. FP %	Std. Dev. TP %
5.52	82.93	3.65	11.19

The qualitative results for four sequenced frames are given in Figure 53. As it can be seen the road regions are classified successfully.

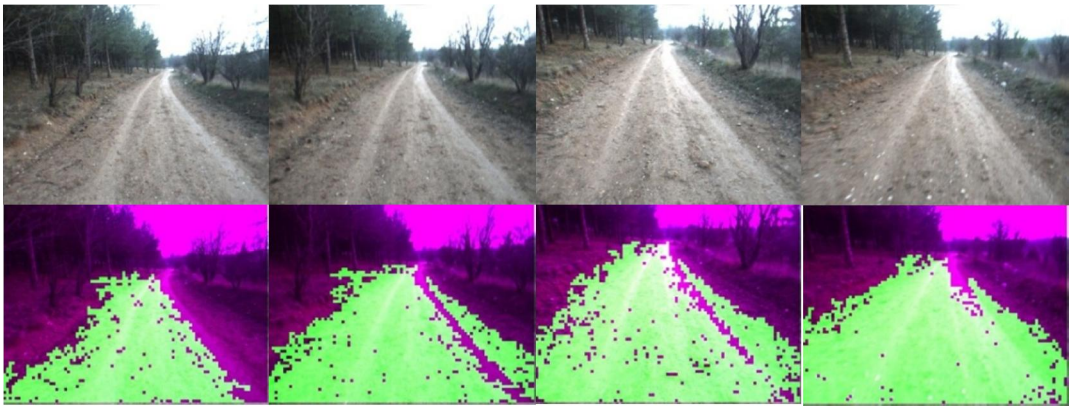

Figure 53 - Four sample sequence and their classification results.

Figure 54 shows the performance of the algorithm under significant light variations. The two sequenced frames with different light conditions are successfully classified. This situation is a direct result of using HSV color space and continuously learning road regions.

Figure 54 - Two sequenced frame with significant light variations. The algorithm can successfully classify the road regions.

3.6 Conclusion

In this chapter, an adaptive self-learning unstructured road detection is presented. The main advantage of the algorithm is to learn the road region by using a simple roughness thresholding feature. The results show that the performance of the algorithm depends on different parameters. Analyzing the effects of these parameters, a parameter set with satisfactory unstructured road detection performance is proposed. The results also show that the algorithm is robust against sudden light changes.

For better classification performance, the height variance threshold should change dynamically with the angle between the camera and ground. However, this will remain as a future work to this academic research.

As mentioned within the chapter, the complexity of the algorithm is high and it cannot be used in real-time. In order to work faster, the algorithm should be implemented in parallel and the parallelization of the algorithm and implementation on a GPU is presented in the next chapter.

CHAPTER 4

CUDA IMPLEMENTATION

4.1 Introduction

In this chapter, CUDA [66] implementation and parallelization of the algorithm proposed in the previous chapter is explained. According to the results and the evaluations given in the previous chapter, it is shown that the algorithm can successfully classify the road regions in the unstructured environments. Despite of its classification success, the time complexity of the algorithm is high and in the current state it is not suitable to run on a robot that requires real-time road detection.

Since there are lots of data to be processed and most of the processing is independent, the algorithm is redesigned in parallel to run the parallel work on the GPU to satisfy real-time considerations. Hence, in this chapter the parallelization of the algorithm as well as the CUDA implementation is presented. The reason of choosing CUDA as the parallel implementation environment is its maturity over other parallel software development environments such as OpenCL [67] or DirectCompute [68]. However, one big disadvantage of using CUDA is that it can only be used for NVIDIA graphics cards. Despite this disadvantage, huge community support that is available for CUDA is a big plus.

In this chapter, a brief introduction for CUDA programming model is given first and then the parallelization of the algorithm with the implementation details are given. Since the implementation details are very important and affect performance severely on GPU computation, implementation of the algorithm is also given.

4.1.1 CUDA Programming Model

With the increase in the demand for real-time, high-definition 3D graphics, graphic processor units, which are the heart of the graphics cards, have evolved into massively parallel manycore processors with tremendous computational power and high memory bandwidth. The computational power as well as the bandwidth of GPUs grew faster than the computational power of CPUs. The grow in the computational power and memory bandwidth in GPUs and CPUs through last decade can be seen in Figure 55 and Figure 56.

Since GPUs are originally designed for graphics rendering which is a compute-intensive, highly parallel computational application, their architecture is developed to handle large data processing instead of data caching and flow control. As a result, more transistors and more chip area are devoted for the computation in GPUs. In this respect, GPU is especially more suitable

for the solution of problems that are data-parallel, that is, if the same program is required to execute for many data elements and the number of memory operations are relatively small. Since memory operations are smaller than the computation in these types of problems, memory access latency can be hidden with calculations instead of big data caches.

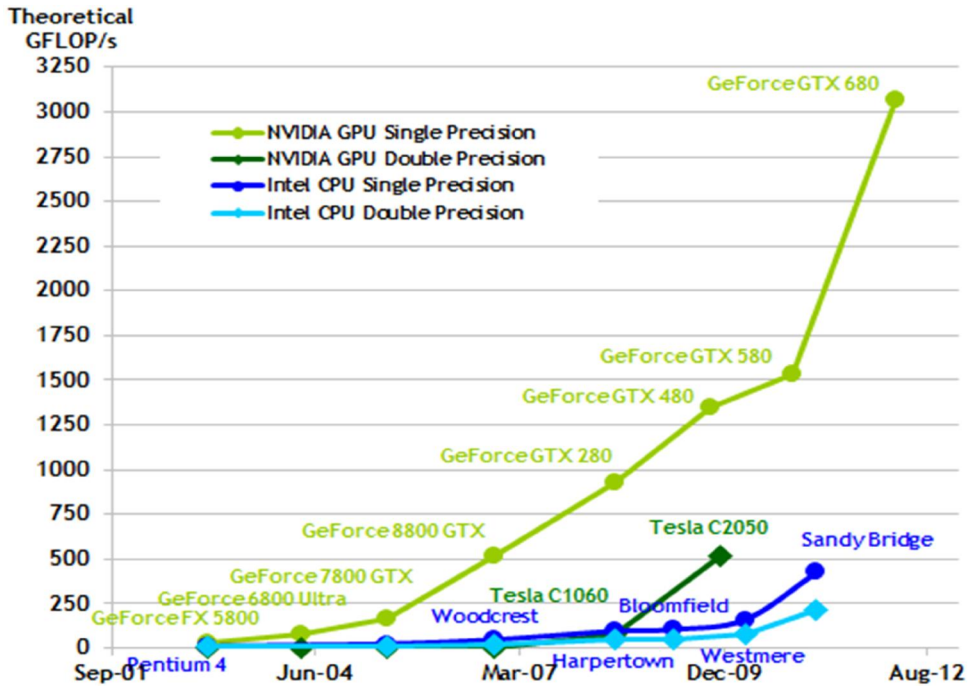

Figure 55 - Floating-Point Operations per Second for the CPU and GPU [66].

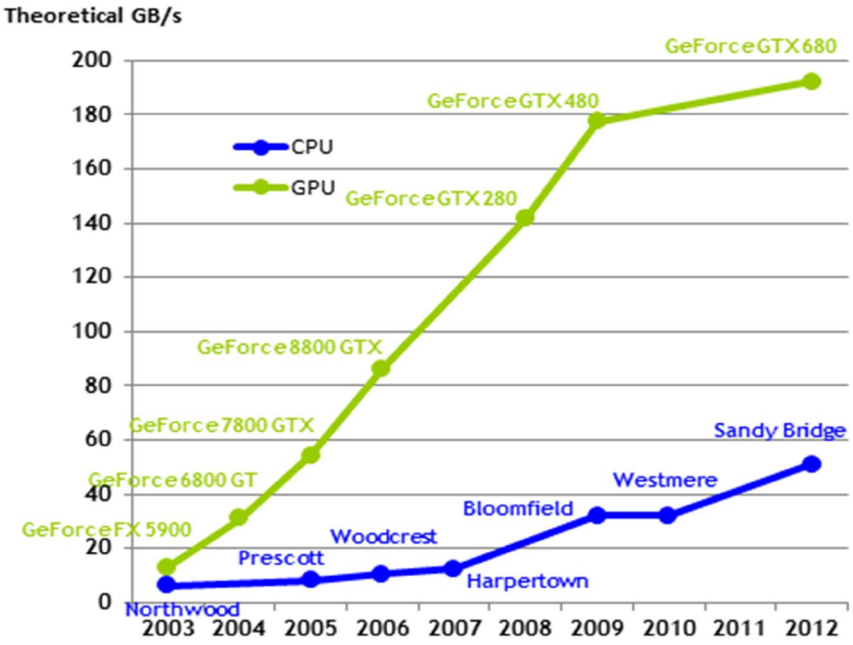

Figure 56 - Memory Bandwidth for the CPU and GPU [66].

In 2006, NVIDIA introduced CUDA [66], a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment (CUDA SDK) that allows developers to use C as a high-level programming language.

The development of multicore CPUs and manycore GPUs puts forward the challenge of developing applications that transparently scales its parallelism to processors with different number of cores. CUDA parallel programming model is designed to beat this challenge and decrease the learning curve of design and implementation of massively parallel applications.

With a minimal set of language extensions done on C language, three core abstractions are done with CUDA: a hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. This decomposition enables automatic scalability between different GPUs with different number of parallel cores.

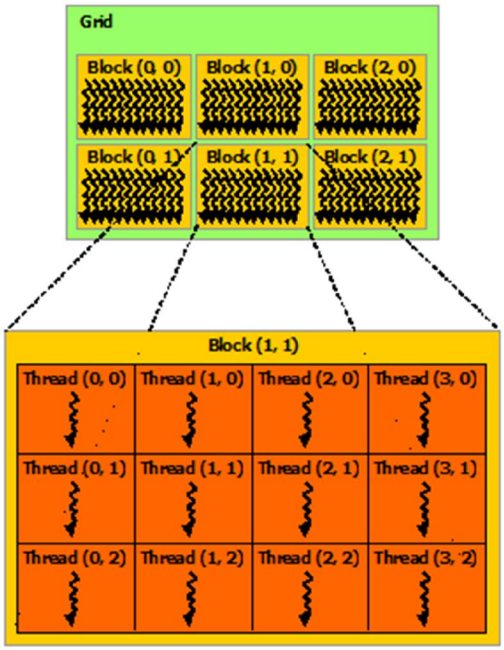

Figure 57 - Thread hierarchy in CUDA programming model [66].

CUDA extends C language by allowing the programmer to define C functions that can be run on GPUs. These functions are called “kernels” and when they called they are executed N times in parallel by N different CUDA threads. Each thread executed knows its unique id through “threadIdx” variable. For example in an image processing implementation, if each thread processes one pixel, then threadIdx variable defines that individual pixel. “threadIdx” is a 3-component vector, so that the index of threads can be one-dimensional, two-dimensional or three-dimensional depending on the application. On the higher hierarchy CUDA programming model has thread blocks. A block is composed of threads and depending on the architecture one block can contain 1024 threads. The thread blocks can be indexed using “blockIdx” variable and can be one-, two- or three-dimensional depending on the GPU architecture. The thread hierarchy is given in Figure 57.

The threads of a thread block execute concurrently on one multiprocessor on GPU hardware and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. These multithreaded processors create, manage, schedule, and execute threads in groups of 32 parallel threads called “warps”. While threads in the same warp are always executed together, warps in the same block are handled asynchronously and, if needed, they can be synchronized. No synchronization is available for all threads in all blocks launched. It can only be done through separate kernel calls.

CUDA programming model also contains a memory hierarchy. Depending on the application, CUDA threads may access data from multiple memory spaces during their execution, namely, registers, local memory, shared memory and global memory. The memory hierarchy is illustrated in Figure 59.

Each thread has registers and private local memory location which can only be reached from the thread itself. The data stored in a register or local memory can only be accessed by the thread itself. Registers are the fastest memory that can be used on the GPU, but has a very limited size per thread. Up to the size of registers, local memory uses registers. However, if a greater size is required, local memory uses global memory, which is the slowest memory space on the card. In the thread block level, each thread block has shared memory which is directly on GPU chip.

Shared memory is visible to all threads of the block and is slower than registers. The total size of the shared memory on the chip is quite limited; 32 kB or 48 kB depending on the architecture. Although slower than registers, shared memory is still much faster than global memory. The main advantage of shared memory space is that the data stored in it can be seen by all threads of a thread block. Thanks to this advantage, cooperative computations such as summation can be done very fast within a thread block. Shared memory space should be used wisely; the use of shared memory greater than the limit will decrease the concurrent thread and thread block execution and degrades the performance severely.

All threads in all blocks have access to the same global memory, which is actually the memory of the graphics card and is not on the GPU chip. Although global memory is the slowest memory space, it is the largest at the same time. Global memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions. In other words, in one transaction a chunk of memory with size 32-, 64-, or 128-bytes is transacted. Thus, to overcome the slowness of the global memory space, the memory access pattern should be coalesced in a warp. If not coalesced, some of the data transacted are not used and decreases efficiency. To achieve a coalesced access pattern, consecutive threads should access the consecutive global memory spaces. A schematic for coalesced access pattern is given in Figure 58.

There are also two read-only memory spaces accessible by all threads, namely, constant and texture memory spaces. Constant memory is cached on the chip and resides in the global memory. It has a limited size. It is fast and can broadcast a reading to a half warp concurrently. Texture memory is also cached on the chip and resides in the global memory. The main advantage of texture memory is that it is optimized for irregular memory access patterns. Thus if the application requires irregular memory accesses, the use of texture memory can improve the performance.

In this thesis work, the high performance paralleling strategy proposed by [69] is applied mainly. According to the strategy proposed by Volkov, to maximize the performance gain the

use of registers should be maximized and independent instructions should be called serially in each kernel as much as possible. Although this strategy might yield more illegible code writing, the increase in the performance is so dramatic that it is usually preferable.

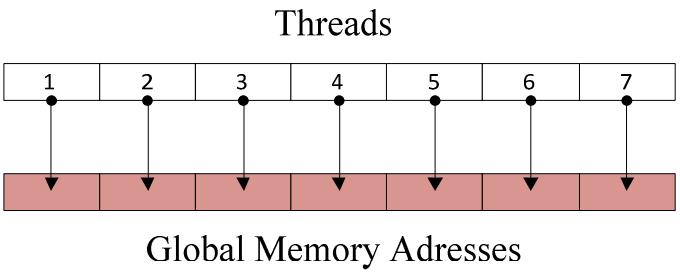

Figure 58 - Schematic representation of coalesced memory access pattern.

4.2 Hardware and Software Environment

Throughout the development of the algorithm all coding is done on a Windows 7 64-bit machine using CUDA C and Visual Studio 2008 [52]. The CUDA implementation is done using NVIDIA CUDA SDK 5.0 [66] and profiling is done on NVIDIA Visual Profiler coming with NVIDIA CUDA Toolkit.

4.3 Parallel Analysis of the Proposed Algorithm

In order to apply GPU acceleration to the solution of a given problem, the problem should be first analyzed to understand its nature. If the problem is not highly parallelizable due to its nature, GPU implementation will not increase the performance of the solution, but severely decrease. In addition to that, while some problems seem serial, they can be expressed and solved with a parallel point of view and parallel approaches work much more efficiently. One example for this type of problem patterns is reduction [70] (e.g. summation) and another is scan [71] (e.g. filtering).

Due to the fact that proposed algorithm is actually composed of several image processing and data mining primitives, the overall nature of the algorithm is massively parallel. As it is expressed in detail in the previous chapter, the algorithm starts with histogram equalization. Since an image is composed of lots of pixels and in histogram equalization these pixels are processed for the computation of image histogram, its equalization and reassigning color values, this part of the algorithm is massively parallel and appropriate to run on a GPU. Due to the same

reason, RGB-HSV conversion is also a massively parallel process and an appropriate work for GPU computation.

Both feature extraction for point cloud data and color data require highly parallel computations which are independent between patches but collaborative within patches. Thus, this part of the proposed algorithm is also suitable for GPU acceleration since GPU thread blocks are designed for collaborative computations. Since filtering of patches according to roughness threshold to generate training samples require a comparison of each patch with the threshold value; this computation can also be implemented on GPU efficiently.

Although connected components labeling algorithm seems not suitable for a parallel processing, there are some efficient implementations on CUDA available, which will be explained in more detail. For large number of training samples, both expectation maximization algorithm and the model updating scheme can be implemented in parallel efficiently while it can be inefficient if the number of training samples are very low. Regular classification with distance calculation and thresholding is also an independent and highly parallel procedure. However, region growing by itself is iterative and in most of these iterations, computations are dependent and not so many. But with a different approach to the problem of classification via region growing, the same problem can be solved with a different and parallelizable manner. This approach is explained in detail in the following chapters.

In the end it can be seen that nearly all of the parts of the proposed algorithm is suitable for parallelization and thus implementation on GPU. Thus, CPUs duty in the algorithm is to move raw data and processed data to/from GPU and controlling the end conditions of iterative parts.

4.4 Algorithm Structure

The algorithm structure of GPU implementation is similar with the CPU version as given in Figure 14. However, this time instead of having point cloud data and color data separately, r,g,b values coming from color image and y coordinate value coming from point cloud data are combined and held in a CUDA data structure float4. Thus, raw data is stored in an array of float4 structures. At the beginning of the algorithm, raw data are copied from CPU memory to GPU memory and bound to texture memory for fast access if irregular global memory access patterns are required.

4.4.1 Data Preprocessing

Data preprocessing part of the algorithm consists of histogram equalization and RGB-HSV conversion as it is given in chapter 3.

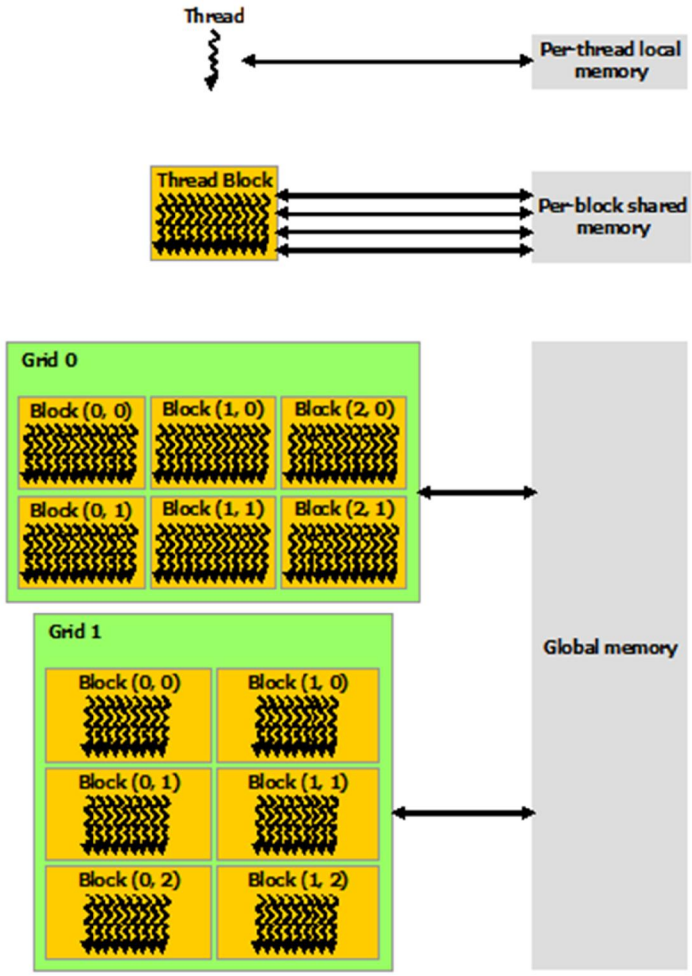

Figure 59 - Memory hierarchy of CUDA programming model [66].

4.4.1.1 Histogram Equalization on GPU

Histogram equalization algorithm consists of four steps [53]:

- Calculation of image histogram $H(I)$
- Normalization of histogram bins to sum 255
- Computation the integral of the histogram $H'_i = \sum_{0 \le j < i} H(j)$
- Transformation of the image using H' as a look up table

The GPU implementation of histogram equalization composed of four kernels:

- Calculation of image histogram

- Calculation of cumulative distribution function (cdf)
- Finding the minimum values in cdf
- Transformation of the image

The first kernel computes the image histograms for each of the RGB channels separately. In order to utilize the parallelization sufficiently, each pixel is assigned to a thread and in the upper level the image is divided into tiles such that each tile is processed by a thread block. The number of threads to run (N_t) and number of blocks to run (N_b) should be selected such that the multiplication of these two ($N_b x N_t$) shoule be a divider of the total number of pixels. Due to the fact that each consecutive threads access to a consecutive global memory address, the global memory transactions are coalesced. Since threads in each thread block can work cooperatively, each thread block has its own sub-histogram part held in shared memory, which is much faster than global memory and can be reached by the threads of a thread block. Then at the end of the kernel, all sub-histograms are summed up and written in the global memory. Histogram calculation kernel is given in Listing 1. The kernel takes r,g,b and y data (d_data) as float4 structure of CUDA and total number of pixels ($size$) as input and returns the histograms of each channel appended to each other as output (d_hist). Thus, red channel histogram is the first 256 value of the histogram (d_hist) while green and blue channel histograms are the second and third 256 values in the array (d_hist). The total shared memory used for each block is equals to the size of r,g and b channel histograms (3x256).

The second kernel calculates the cumulative distribution functions (cdf) of the histograms of each channel by using parallel sum scan algorithm [71] and is described in [72]. A sum scan algorithm basically calculates the sum of all elements up to an element of an array from the beginning of the array. A naïve sum scan algorithm can be given as in Figure 60 schematically.

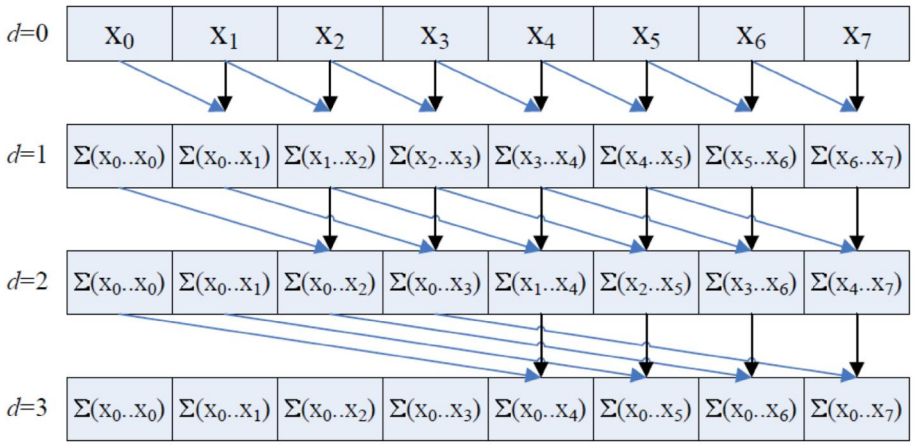

Figure 60 - A schematic of the naive sum scan algorithm [72].

If the last element of the array is included in the resultant array, it is called an inclusive scan and called an exclusive scan otherwise. The algorithm described in [72] is an exclusive scan algorithm, but for cdf calculation it is converted to an inclusive scan algorithm. The algorithm is also modified to calculate the cdf of all three channels. Since number of bins for image histogram is 256, the kernel is run with 1 thread block consisting of 128 threads. The number of threads for this implementation is required to be half of the number of bins and the shared memory allocated should be 3*number_of_bins+1 since there are three channels. The kernel requires the original histograms (d_hist), number of bins (n_bins) and total number of pixels ($size\_max$) as input and returns the cdfs of each channel appending to each other as output (d_cdf) as given in Listing 2. Thus, the first 256 values are the cdf of red channel while the second and third 256 values are the cdfs of green and blue channels in the array.

```
function kernel_hist(d_data, d_hist, size)
        declare uint tempr, tempg, tempb in shared memory
        declare int i ← threadIdx.x + blockIdx.x * blockDim.x
        declare int offset ← blockDim.x * gridDim.x

        if threadIdx.x < 256 then
                tempr[threadIdx.x] ← 0
                tempg[threadIdx.x] ← 0
                tempb[threadIdx.x] ← 0
        end if
        synchronize block

        repeat
                declare float4 dummy ← d_data[i]
                atomicAdd(tempr[dummy.x],1)
                atomicAdd(tempg[dummy.x],1)
                atomicAdd(tempb[dummy.x],1)
                i ← i + offset
        until i >= size
        synchronize block

        if threadIdx.x < 256 then
                atomicAdd(d_hist[threadIdx.x],tempr[threadIdx.x])
                atomicAdd(d_hist[threadIdx.x+256],tempg[threadIdx.x])
                atomicAdd(d_hist[threadIdx.x+512],tempb[threadIdx.x])
        end if
        return
```

Listing 1 - Histogram calculation kernel.

```
function kernel_prefixsumscan_cdf(d_cdf, d_hist, n_bins, size_max)
    declare uint temp in shared memory
    declare int thid ← threadIdx.x
    declare int offset ← 1
    declare int ai ← thid
    declare int bi ← thid + (n_bins/2)
    declare int bankOffsetA ← CONFLICT_FREE_OFFSET(ai)
    declare int bankOffsetB ← CONFLICT_FREE_OFFSET(bi)
    declare int cnst ← n_bins + CONFLICT_FREE_OFFSET(n_bins - 1)

    temp[ai + bankOffsetA] ← d_hist[ai]
    temp[bi + bankOffsetB] ← d_hist[bi]
    temp[cnst + ai + bankOffsetA] ← d_hist[n_bins  + ai]
    temp[cnst + bi + bankOffsetB] ← d_hist[n_bins  + bi]
    temp[2 * cnst + ai + bankOffsetA] ← d_hist[2 * n_bins  + ai]
    temp[2 * cnst + bi + bankOffsetB] ← d_hist[2 * n_bins  + bi]

    declare d ← n_bins >> 1
    repeat
        synchronize block
        if thid < d then
            ai ← offset*(2*thid+1)-1
            bi ← offset*(2*thid+2)-1
            ai ← ai + CONFLICT_FREE_OFFSET(ai)
            bi ← bi + CONFLICT_FREE_OFFSET(bi)
            temp[bi] ← temp[bi] + temp[ai]
            temp[cnst+bi] ← temp[cnst+bi] + temp[cnst+ai]
            temp[2*cnst+bi] ← temp[2*cnst+bi] + temp[2*cnst+ai]
        end if
        offset ← offset * 2
        d ← d >> 1
    until d <= 0

    if thid = 0 then
        temp[cnst - 1] ← 0
        temp[2 * cnst - 1] ← 0
        temp[3 * cnst - 1] ← 0
    end if
```

```
d ← 1
repeat
        offset ← offset / 2
        synchronize block
        if thid < d then
                ai ← offset*(2*thid+1)-1
                bi ← offset*(2*thid+2)-1
                ai ← ai + CONFLICT_FREE_OFFSET(ai)
                bi ← bi + CONFLICT_FREE_OFFSET(bi)
                declare uint t ← temp[ai]
                declare uint t2 ← temp[cnst+ai]
                declare uint t3 ← temp[2*cnst+ai]
                temp[ai] ← temp[bi]
                temp[ai+cnst] ← temp[bi+cnst]
                temp[ai+2*cnst] ← temp[bi+2*cnst]
                temp[bi] ← temp[bi] + t
                temp[bi+cnst] ← temp[bi+cnst] + t2
                temp[bi+2*cnst] ← temp[bi+2*cnst] + t3
        end if
until d >= n_bins
synchronize block

if ai != 0 then
        d_cdf[ai−1] ← temp[ai+bankOffsetA]
        d_cdf[n_bins+ai−1] ← temp[cnst+ai+bankOffsetA]
        d_cdf[2*n_bins+ai−1] ← temp[2*cnst+ai+bankOffsetA]
else
        d_cdf[n_bins−1] ← size_max
        d_cdf[2*n_bins−1] ← size_max
        d_cdf[3*n_bins−1] ← size_max
end if
d_cdf[bi−1] ← temp[bi+bankOffsetB]
d_cdf[n_bins+bi−1] ← temp[cnst+bi+bankOffsetB]
d_cdf[2*n_bins+bi−1] ← temp[2*cnst+bi+bankOffsetB]
return
```

Listing 2 - Cdf calculation kernel based on [72].

```
function kernel_findmin(d_cdf, d_min, n_bins)
        declare uint s_cdf in shared memory
        s_cdf[threadIdx.x] ← d_cdf[threadIdx.x]
        s_cdf[n_bins+threadIdx.x] ← d_cdf[n_bins+threadIdx.x]
        s_cdf[2*n_bins+threadIdx.x] ← d_cdf[2*n_bins+threadIdx.x]

        declare s ← n_bins >> 1
        repeat
                if thid < s then
                        declare float dr1 ← s_cdf[threadIdx.x]
                        declare float dr2 ← s_cdf[threadIdx.x+s]
                        declare float dg1 ← s_cdf[n_bins+threadIdx.x]
                        declare float dg2 ← s_cdf[n_bins+threadIdx.x+s]
                        declare float db1 ← s_cdf[2*n_bins+threadIdx.x]
                        declare float db2 ← s_cdf[2*n_bins+threadIdx.x+s]

                        if dr1 = 0 or dr2 = 0  then
                                s_cdf[threadIdx.x] ← max(dr1,dr2)
                        else
                                s_cdf[threadIdx.x] ← min(dr1,dr2)
                        end if
                        if dg1 = 0 or dg2 = 0  then
                                s_cdf[n_bins+threadIdx.x] ← max(dg1,dg2)
                        else
                                s_cdf[n_bins+threadIdx.x] ← min(dg1,dg2)
                        end if
                        if db1 = 0 or db2 = 0  then
                                s_cdf[2*n_bins+threadIdx.x] ← max(db1,db2)
                        else
                                s_cdf[2*n_bins+threadIdx.x] ← min(db1,db2)
                        end if
                end if
                synchronize block
                s ← s >> 1
        until s <= 0
        if threadIdx.x = 0 then
                dmin ← uint4(s_cdf[threadIdx.x], s_cdf[n_bins+threadIdx.x],
                        s_cdf[2*n_bins+threadIdx.x], 0)
        end if
        return
```

Listing 3 - Kernel to find minimum values of cdfs.

The third kernel finds the minimum values other than zero in the cumulative density functions (d_cdf) of each channel with a standard parallel sum reduction [70] as given in Figure 61. Since cdfs has 256 bins, the CUDA kernel is run with 256 threads and a single thread block, and the shared memory size used is the three times of one channel's histogram size (3*256). In order this kernel to be used, the number of bins of the cdfs should be a power of two due to the nature of parallel reduction. The algorithm takes cdf of all three channels (d_cdf) and number of bins in a cdf (n_bins) as input and returns the minimum value of cdfs as a uint4 structure (d_min). The kernel implementation is given in Listing 3.

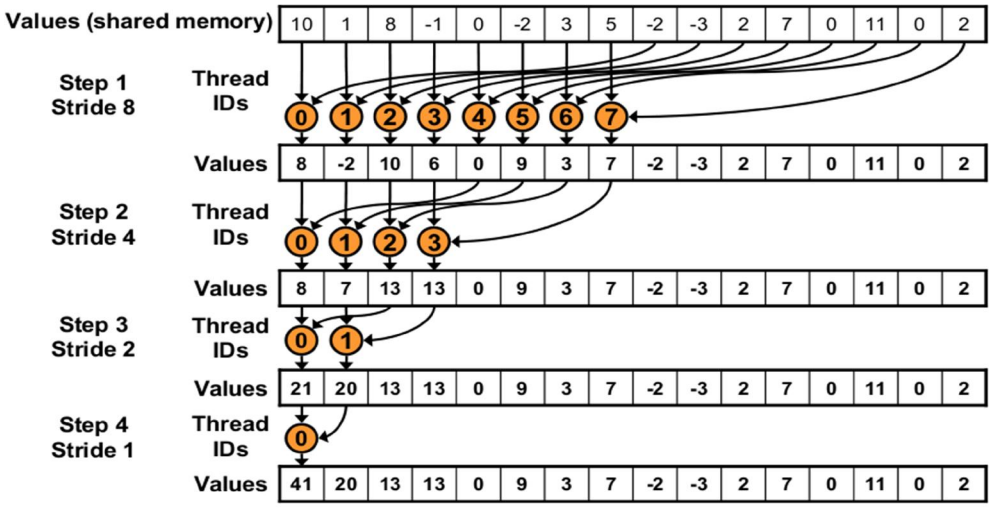

Figure 61 - Parallel sum reduction tree [70].

The fourth kernel is given in Listing 4. The kernel basically rescales the red, green and blue values according to the cdf functions and minimum value in the cdfs found for each channel. While rescaling the pixel values, the global memory storing cdfs is accessed in a very irregular pattern and the performance of the algorithm degrades severely. To remove this degradation and increase the performance, the memory on the GPU's ram that stored cdfs is bound to texture memory before this kernel launched. Since texture memory has a better performance in this type of irregular memory access patterns, it increases the performance. As it is the case in the first kernel, in this kernel each pixel is assigned to one thread and the whole image is divided into tiles and each tile is assigned to one thread block. The number of threads to run (N_t) and number of blocks to run (N_b) should be selected such that the multiplication of these two ($N_b x N_t$) should be a divider of the total number of pixels. The kernel takes r,g,b and y data as float4 structure (d_data), cdfs (d_cdf), minimum values of cdfs (d_min), total number of pixels

($size$) and number of bins (n_bins) in an image histogram as input and writes the rescaled image data on the old r, g, b data (d_data).

```
function kernel_histeq(d_data,  d_min, n_bins, size)
        declare int i ← threadIdx.x + blockIdx.x * blockDim.x
        declare int offset ← blockDim.x * gridDim.x
        repeat
                declare float4 dummynew
                declare float4 dummy ← d_data[i]
                declare uint dcdfr ← texfetch (d_cdf, dummy.x)
                declare uint dcdfg ← texfetch (d_cdf, n_bins+dummy.y)
                declare uint dcdfb ← texfetch (d_cdf, 2*n_bins+dummy.z)
                dummynew.x ← round((dcdfr-min.x)*255f/(size-1))
                dummynew.y ← round((dcdfg-min.y)*255f/(size-1))
                dummynew.z ← round((dcdfb-min.z)*255f/(size-1))
                dummynew.w ← dummy.w
                d_data[i] ← dummynew
                i ← i + offset
        until i >= size
        return
```

Listing 4 - Kernel to transform the image to histogram equalized image using cdfs.

4.4.1.2 RGB-HSV Conversion

The procedure for RGB-HSV conversion is given in [53] for CPU and on GPU this procedure is taken as base. In Listing 5, its GPU implementation is given. In this implementation, again each pixel is assigned to a pixel and the whole image is divided into tiles and each tile is assigned to a thread block to get enough parallelization for performance increase and achieve a coalesced global memory access pattern. The number of threads to run (N_t) and number of blocks to run (N_b) should be selected such that the multiplication of these two ($N_b x N_t$) shoule be a divider of the total number of pixels. The algorithm takes r,g,b and y data as float4 structure (d_data) and total number of pixels ($size$) as input and writes HSV image data on the old r, g, b data (d_data).

```
function kernel_rgb2hsv(d_data,  size)
        declare int i ← threadIdx.x + blockIdx.x * blockDim.x
        declare int offset ← blockDim.x * gridDim.x
        repeat
                declare float4 dummy ← d_data[i]
                dummy.x ← dummy.x/255f
                dummy.y ← dummy.y/255f
                dummy.z ← dummy.z/255f
                declare float max ← max(dummy.x, dummy.y, dummy.z)
                declare float min ← min(dummy.x, dummy.y, dummy.z)
                declare float delta ← max – min
                declare float s ← delta/max
                synchronize block
                declare float f
                if max = 0 or s = 0  then h ← 0
                else
                        if dummy.x = max then
                                h ← (dummy.y-dummy.z)/delta
                        else if dummy.y = max then
                                h ← 2f+(dummy.z-dummy.x)/delta
                        else if dummy.z = max then
                                h ← 4f+(dummy.x-dummy.y)/delta
                        end if
                end if
                synchronize block
                h ← h * 60
                if h < 0 then h ← h + 360
                dummy.x ← h / 2f
                dummy.y ← 255f * s
                dummy.z ← 255f * max
                d_data[i] ← dummy
                i ← i + offset
        until i >= size
        return
```

Listing 5 - RGB-HSV conversion kernel.

```
function kernel_feature_extraction(d_data,  d_hists, d_trlabels,d_trsize, d_ntr,
width, npatches, sizepatch, threshold_var_y)
        declare int pid ← threadIdx.x + blockIdx.x * blockDim.x
        declare int offset ← blockDim.x * gridDim.x
        declare int binSat ← 256 / N_BINS
        declare int binHue ← 181 / N_BINS
        declare uint lhist in local memory
        repeat
                declare uint prow ← pid/(width/sizepatch)
                declare uint pcol ← pid%(width/sizepatch)
                declare uint basepixid ← prow*sizepatch*width+pcol*sizepatch
                declare float y ← 0
                declare float y_sqr ← 0
                initialize lhist to zero
                for all pixel in patch do
                        declare char hue ← pixel.x/binHue
                        declare char sat ← pixel.y/binHue
                        lhist[hue*N_BINS+sat] ← lhist[hue*N_BINS+sat]+1
                        y ← y + pixel.w
                        y_sqr ← y_sqr + pixel.w * pixel.w
                end for
                declare i ← 0
                repeat
                        d_hist[i*n_patches+pid] ← lhist[i]
                until i >= N_BINS_SQR
                declare var ← (y_sqr – (y*y)/(sizepatch^2)/ (sizepatch^2-1))
                if var != 0 or var < threshold_var_y then
                        d_trlabels[pid] ← 1
                        d_trsize[pid] ← 1
                        atomicAdd(d_ntr, 1)
                end if
                pid ← pid + offset
        until pid >= npatchsize
        return
```

Listing 6 - Kernel implementation for feature extraction with patch filtering.

4.4.2 Feature Extraction and Automatic Training Sample Generation

This part can be divided into two: Feature extraction with patch filtering according to roughness and finding the biggest connected sample group. While feature extraction for patches and patch

filtering according to roughness threshold are unified in a single kernel, due to its iterative and quite complex nature, connected components labeling and the search for the biggest connected training sample group are composed of several kernels.

4.4.2.1 Feature Extraction with Patch Filtering

In order to remove the burden of running several kernels for point cloud feature extraction, color feature extraction and filtering patches according to roughness threshold, they are unified in a single kernel. With the removal of the burden, increase the performance is aimed. As roughness feature height variance is used for point cloud patches.

As it is given in Listing 6, the algorithm takes r,g,b and y coordinate data as float4 structure (d_data), width of the image ($data$), number of patches to be generated ($npatches$), size of a patch ($sizepatch$) in pixels and height variance threshold ($threshold\_var\_y$) as input and returns H-S joint histograms of patches (d_hists), a binary downsampled image ($d\_trlabels$) that marks the locations of training samples or patches, and a copy of this binary image that will be used to calculate the size of each connected component (d_trsize), and the number of total training samples gathered as outputs (d_ntr). The binary downsampled image ($d\_trlabels$) will be used to hold the labels in the connected components labeling algorithm later on. The binary downsampled image ($d\_trlabels$) and its clone to be used in size calculation (d_trsize) are initialized to 0 at the global memory allocation. Since the height variance is only used in the algorithm for gathering training samples, height variance features are not stored in the memory.

This time each thread is assigned to a patch and the patch image (downsampled image) is divided into tiles. Each tile is assigned to a thread block to achieve optimum parallelization with coalesced global memory access. The number of threads to run (N_t) and number of blocks to run (N_b) should be selected such that the multiplication of these two ($N_b x N_t$) should be a divider of the total number of patches. Since registers are faster than any memory available, each thread holds its own H-S joint histogram, y coordinate sum and y coordinate squared sums on registers to have fast computation and at the end these computed histograms are stored in the global memory. At the end of the kernel by using calculated variance of height, the training samples are marked in the two binary images ($d\_trlabels$ and d_trsize). Thus if one patch's height variance is smaller than the threshold it is marked as 1, else the memory location left at 0.

4.4.2.2 Finding the Biggest Connected Sample Group

This part is composed of six kernels in total. Two of them are for connected components labeling algorithm and the remaining four are used to finding the biggest connected component group of samples and moving the training samples to a separate memory.

4.4.2.2.1 Connected Components Labeling

As it is mentioned, connected components labeling algorithm is composed of two kernels. One of these kernels is just for the initialization of labels on the binary downsampled image and the

other is for the actual connected component labeling part. While the first kernel is called only once at the beginning of this part, the second kernel is called iteratively until there are no labels propagating. In this study, one of the GPU accelerated connected components labeling algorithm explained by Hawick et al. in [73] is used with minor adaptations as given in [74]. A connected components labeling algorithm performing local neighbor propagation, named as *Mesh_Kernel_B* in [73], is used in the implementation. Note that this algorithm is not the most efficient one but its performance is enough when its simplicity is considered.

Initialization kernel is a simple kernel as given in Listing 7. It takes the binary downsampled matrix ($d\_trlabels$) that marks the locations of training samples created in the feature extraction kernel and the total number of patches ($size$) as inputs and initialize the labels and write them on the downsampled matrix ($d\_trlabels$). The whole matrix is divided into tiles and each tile is assigned to a thread block while each patch is assigned to a thread. The number of threads to run (N_t) and number of blocks to run (N_b) should be selected such that the multiplication of these two ($N_b x N_t$) should be a divider of the total number of patches. If the value on the downsampled matrix ($d\_trlabels$) is 1, the global thread id is assigned to that patch as the initial label; otherwise the label remains as 0. As it can be seen in Listing 7, this implementation is done without using an if clause. Elimination of "if" clause removes branch divergence in warps and results in a performance increase up to 2 times. Since each consecutive thread reads and writes to a consecutive global memory address, the global memory transactions are coalesced.

```
function kernel_initLabels(d_trlabels, size)
        declare int gid ← threadIdx.x + blockIdx.x * blockDim.x
        declare int offset ← blockDim.x * gridDim.x
        repeat
                d_trlabels[gid] ← d_trlabels[gid] * gid
                gid ← gid + offset
        until gid >= size
        return
```

Listing 7 - Kernel implementation for the initialization of connected components labeling algorithm.

The local neighbor propagation kernel is given in Listing 8. The whole patch image ($d\_trlabels$) that contains the initial labels of training samples is divided into tiles. Each thread block is assigned to a tile and each thread is assigned to one patch. The size of shared memory used in each block is equal to the number of threads in each thread block.

The kernel requires the image that is initialized in the previous kernel ($d\_trlabels$), the copy of the binary image that holds training sample locations (d_trsize), the number of tiles in the width and height of image ($widthgrid$ and $heightgrid$) as inputs and returns the labels of connected components on the input binary patch image ($d\_trlabels$), the size of each connected component (d_trsize) and a Boolean ($m\_d\_IsNotDone$) that tracks if there is a change in the labels in the given iteration.

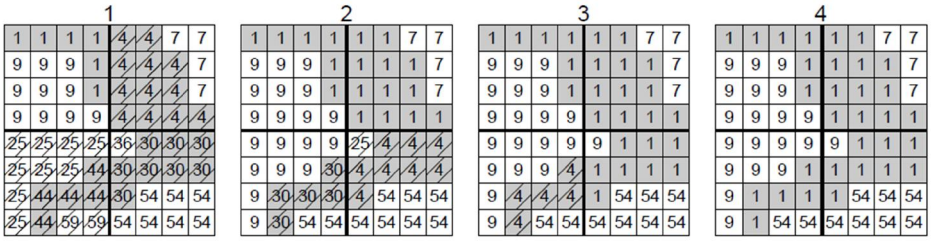

Figure 62 - Four iterations of the local label propagation algorithm. In this case the image is divided into four blocks indicated by heavy lines. Patches that are crossed out are yet to receive their final label [73].

Initially the threads fetch their patch's label and the labels of all four neighbors from global memory and choose the lowest label of these. Then, the threads write these values into shared memory and the kernel loops through updating the labels within shared memory until they stop changing. In each iteration of this loop, the neighbors' labels are loaded from shared memory and the lowest appropriate label is chosen. This effectively performs the entire labeling algorithm on each block at a time.

The kernel also writes the number of patches in each connected component to the place in the binary copy (d_trsize) where the smallest label is. As the labels on the borders of the block will change, multiple calls of this kernel are required to correctly label the entire image. Hence, the kernel iterates until there is no change in the labels. The schematic representation of the algorithm is given in Figure 62.

```
function kernel_ccl_mesh_B_improved(d_trlabels, d_trsize, m_d, widthgrid,
heightgrid, size)
        declare int nid in local memory
        declare int Ll, ml in shared memory
        declare int id ← (threadIdx.y + blockIdx.y * blockDim.y) * widthgrid +
threadIdx.x + blockIdx.x * blockDim.x
        declare int idl ← threadIdx.y * blockDim.x + threadIdx.x
        declare int idn ← 0
        declare uint label ← texfetch(d_trlabels, id)
        ml ← 1
        if label != 0 then
                if (id - widthgrid) >= 0 and label * texfetch (d_trlabels, id-
widthgrid)
                        nid[idn] ← texfetch (d_trlabels, id-widthgrid)
                        idn ← idn + 1
                end if
                if (id + widthgrid) < widthgrid * heightgrid and label *
tex1Dfetch(d_trlabels, id+widthgrid)
                        nid[idn] ← texfetch (d_trlabels, id+widthgrid)
                        idn ← idn + 1
                end if
                if ((id - 1) % widthgrid) != widthgrid - 1 and label *
        tex1Dfetch(d_trlabels, id-1)
                        nid[idn] ← texfetch (d_trlabels, id-1)
                        idn ← idn + 1
                end if
                if ((id + 1) % widthgrid) != 0 and label * texfetch (d_trlabels,
        id+1)
                        nid[idn] ← texfetch (d_trlabels, id+1)
                        idn ← idn + 1
                end if
                declare int i ← 0
                repeat
                        if nid[i] < label then
                                label ← nid[i]
                                md ← 1
                        end if
                i ← i + 1
                until i >= idn
```

```
    idn ← 0
    if (idl–blockDim.x)>=0 and label* texfetch (d_trlabels, id-widthgrid)
            nid[idn] ← idl - blockDim.x
            idn ← idn + 1
    end if
    if (idl+blockDim.x)<blockDim.x*blockDim.y and label*texfetch
(d_trlabels, id+widthgrid)
            nid[idn] ← idl + blockDim.x
            idn ← idn + 1
    end if
    if ((idl - 1) % blockDim.x) != blockDim.x - 1 and label * texfetch
(d_trlabels, id-1)
            nid[idn] ← idl - 1
            idn ← idn + 1
    end if
    if ((idl+1)%blockDim.x) != 0 and label * texfetch (d_trlabels, id+1)
            nid[idn] ← idl + 1
            idn ← idn + 1
    end if
    repeat
            Ll[idl] ← label
            synchronize block
            ml ← 0
            i ← 0
            repeat
                    if Ll[nid[i]] < label then
                            label ← Ll[nid[i]]
                            ml ← 1
                    end if
            synchronize block
            i ← i + 1
            until i >= idn
            d_trlabels[id] ← label
            declare uint n ← d_trsize[ id]
            if n && label != id then
                    atomicAdd(d_trsize[label], n)
                    d_trsize[id] ← 0
            end if
        until ml != true
end if
return
```

Listing 8 - The local neighbor propagation kernel.

```
function kernel_findmax(d_trsize, d_max, d_max_gid, size_max)
        declare uint s_max,s_max_gid in shared memory
        declare uint gid ← threadIdx.x
        declare uint max ← 0
        declare uint max_gid ← 0
        repeat
                s_max [threadIdx.x] ← d_trsize[gid]
                s_max_gid [threadIdx.x] ← gid
                declare s ← blockDim.x >> 1
                repeat
                        if threadIdx.x < s then
                                declare uint dum1 ← s_max[threadIdx.x]
                                declare uint dum2 ← s_max[threadIdx.x+s]
                                if dum1 < dum2 then
                                        s_max[threadIdx.x] ← dum2
                                        s_max_gid[threadIdx.x] ←
                                s_max_gid[threadIdx.x+s]
                                end if
                        end if
                        synchronize block
                        s ← s >> 1
                until s <= 0
                if threadIdx.x = 0 and s_max[threadIdx.x] > max then
                        max ← s_max[threadIdx.x]
                        max_gid ← s_max_gid[threadIdx.x]
                end if
                synchronize block
                gid ← gid + blockDim.x
        until gid >= size_max
        if threadIdx.x = 0 then
                        d_max[threadIdx.x] ← max
                        d_max_gid[threadIdx.x] ← max_gid
        end if
        return
```

Listing 9 - Kernel implementation of finding the largest connected component location.

4.4.2.2.2 Selection of the Biggest Sample Group

The first kernel in this part is the kernel for the search of the largest connected component and the number of patches in this component. Finding the maximum in an array or matrix is a

parallel reduction problem (but not parallel sum this time). The whole downsampled image that holds the size of the connected components (d_trsize) is assigned to one thread block and each element of it is assigned to a thread. This kind of parallelization yields a coalesced global memory access and increase performance. The implementation uses shared memory and the size of the shared memory should be equal to the number of threads in the thread block. Since the size of the image is relatively small, even with one thread block the computation takes very short time. Thus, the implementation is remained with one thread block. However, the number of threads in the thread block should be a power of two due to the nature of reduction pattern. The kernel takes the image storing sizes of each connected component computed in the previous kernel (d_trsize) and the total number of patches ($size\_max$) in the image as inputs and returns the memory location of ($d\_\max\_gid$) and the number of training patches (d_max) in the biggest connected component. The kernel implementation is given in Listing 9.

The next kernel is a simple kernel that filters the biggest connected component and creates a new binary image of patches where if a patch belongs to the biggest connected component, its value is 1 and 0 otherwise. This new binary image is written on the input patch image ($d\_trlabels$) and is used to find the global memory locations of the patches belonging to the largest connected component in the next kernel. The patch image is again divided into tiles and each tile is assigned to one thread block while each patch is assigned to a thread. The number of threads to run (N_t) and number of blocks to run (N_b) should be selected such that the multiplication of these two ($N_b x N_t$) should be a divider of the total number of patches. The kernel implementation is given in Listing 10. The kernel takes label image ($d\_trlabels$) or image generated with connected components labeling, the memory location of the biggest connected component (d_maxgid) and the number of total patches in an image ($n\_patches$) as inputs and writes a binary image that only contains the patches in the biggest connected component on the downsampled label image ($d\_trlabels$).

After filtering the patches in the biggest connected component, another kernel is run to determine the memory index where each of the training sample patches is stored in the training buffer. Although for a CPU implementation such an implementation is trivial since CPUs run in a serial fashion, due to the fact that GPUs are running parallel and concurrently finding memory indices to write is quite complicated, but can be implemented parallel and fast. This application is called parallel compact and used often in filtering applications such as scene rendering. Since the locations of the patches belonging to the biggest connected component are marked with 1 and 0 otherwise, a parallel prefix sum scan pattern [71] can be applied as it is applied to find the cumulative distribution function of image histogram. Different from the cdf implementation, the parallel prefix sum scan algorithm required here is an exclusive scan algorithm same with [72]. The kernel takes the binary image generated in the previous kernel call ($d\_trlabels$) and the total number of patches in the image ($size\_max$) as inputs and returns an image ($d\_trcompact$) with the same size of the total number of image patches but consist the memory indices where the training samples will be written in the training buffer. One thread block is assigned to the whole patch matrix. The kernel is also required that number of patches should be divisible to

twice of the number of threads in the block. Number of threads should be a power of two. The kernel implementation is given in Listing 11.

```
function kernel_filterbiggestblob(d_trlabels, d_max_gid, n_patches)
        declare uint gid ← blockIdx.x * blockDim.x + threadIdx.x
        declare uint max ← d_max_gid[0]
        repeat
                if d_trlabels[gid] = max then
                        d_trlabels[gid] ← 1
                else
                        d_trlabels[gid] ← 0
                end if
                gid ← gid + blockDim.x * gridDim.x
        until gid >= n_patches
        return
```

Listing 10 - Kernel implementation for biggest connected component filtering.

The last kernel implementation of this section is a simple kernel used to put only the training samples into the training buffer. The patch image that contains the memory indices of training buffer ($d\_trcompact$) is divided into tiles and each tile is assigned to a block of threads. Each patch, thus, assigned to a thread. The number of threads to run (N_t) and number of blocks to run (N_b) should be selected such that the multiplication of these two ($N_b x N_t$) should be a divider of the total number of patches. The kernel basically checks whether there are any training samples in the patch location. If there is a training sample, then using the memory index found in the previous kernel, the feature vector related to that sample is inserted into that training memory index. As a result of this kernel the feature vectors of training samples are stored consecutively in the training buffer ($d\_trbuffer$).

The kernel takes the binary image contains only the biggest connected component ($d\_trlabels$), the training buffer indices of training samples ($d\_trcompact$), H-S joint histograms of patches calculated previously (d_hists), the number of patches in the largest connected component (d_max), and the total number of patches in an image ($n\_patches$) as inputs and returns the training memory containing selected feature vectors as output ($d\_trbuffer$). The implementation of the kernel is given in Listing 12.

```
function kernel_prefixsumscan_compact_uint(d_trcompact, d_trlabels,
size_max)
        declare uint sum, temp in shared memory
        declare int gid ← threadIdx.x
        declare int goffset ← 2 * blockDim.x
        sum ← 0
        repeat
                declare int thid ← threadIdx.x
                declare int offset ← 1
                declare int ai ← thid
                declare int bi ← thid + blockDim.x
                declare bankOffsetA ← CONFLICT_FREE_OFFSET(ai)
                declare bankOffsetB ← CONFLICT_FREE_OFFSET(bi)
                temp[ai+ bankOffsetA] ← texfetch (d_trlabels, gid)
                temp[bi+ bankOffsetB] ← texfetch (d_trlabels, gid + goffset/2)
                declare d ← blockDim.x
                repeat
                        synchronize block
                        if thid < d then
                                ai ← offset*(2*thid+1)-1
                                bi ← offset*(2*thid+2)-1
                                ai ← ai + CONFLICT_FREE_OFFSET(ai)
                                bi ← bi + CONFLICT_FREE_OFFSET(bi)
                        end if
                        offset ← offset * 2
                        d ← d >> 1
                until d <= 0
                if thid = 0 then
                        temp[2*blockDim.x-1 +
                CONFLICT_FREE_OFFSET(2*blockDim.x-1)] ← sum[0]
                end if
```

```
        d ← 1
        repeat
            offset ← offset / 2
            synchronize block
            if thid < d then
                ai ← offset*(2*thid+1)-1
                bi ← offset*(2*thid+2)-1
                ai ← ai + CONFLICT_FREE_OFFSET(ai)
                bi ← bi + CONFLICT_FREE_OFFSET(bi)
                declare uint t ← temp[ai]
                temp[ai] ← temp[bi]
                temp[bi] ← temp[bi] + t
            end if
            d ← d * 2
        until d >= 2*blockDim.x
        synchronize block
        d_trcompact[gid] ← temp[ai+bankOffsetA]
        d_trcompact[gid+goffset/2] ← temp[bi+bankOffsetB]
        if thid = blockDim.x - 1 then
            sum ← temp[bi+ bankOffsetB] + texfetch (d_trlabels, gid +
goffset/2)
        gid ← gid + offset
until gid >= size_max
return
```

Listing 11 - Kernel implementation of exclusive prefix sum scan algorithm to find the indices for the patches belonging to the biggest connected component.

4.4.3 Color Distribution Model Learning and Updating

The logic of color distribution model learning and the update of the learned color models are essentially the same as the CPU implementation, but parallelized.

4.4.3.1 Model Learning

After training samples obtained in a separate array ($d\_trbuffer$), by using these training samples the Gaussian mixture models are learned through parallel expectation maximization algorithm. The parallel expectation maximization algorithm used in this study is the one proposed by Pangborn in his master thesis [75]. In short, new cluster memberships are computed for each event in the E-step using Gaussian model parameters from the previous iteration, and then the parameters are updated using the new memberships in the M-step. Iteration continues until the total change in likelihood (which is computed in the E-step) is less than some user-

specified epsilon value or the number of iterations is greater than some maximum iteration criterion.

```
function kernel_insert_training_samples_w_inds(d_trlabels, d_trcompact,
d_hists, d_trbuffer, d_trinds, d_max, n_patches)
        declare int gid ← threadIdx.x + blockIdx.x * blockDim.x
        declare int offset ← blockDim.x * gridDim.x
        declare uint ntr ← d_max
        repeat
                if d_trlabels[gid] != 0 then
                        declare int id ← d_trcompact[gid]
                        declare int i ← 0
                        repeat
                                d_trbuffer[i*n_tr+id] ← d_hists[i*n_patches+gid]
                                i← i+1
                        until i >= N_BINS_SQR
                        d_trinds[id] ← 1
                end if
                gid ← gid + offset
        until gid >= n_patches
        return
```

Listing 12 – Kernel implementation for inserting training samples (the biggest connected component) to a separate memory.

Similar to the CPU implementation, the maximum number of iterations is set to 100. The main difference between the work of Pangborn and the parallel EM implementation given here is that Pangborn's implementation is for a cluster of GPUs while the implementation in this study is for a single GPU. Since Pangborn's work requires the processing of much much larger data than in this thesis work, the researcher's implementation is for a cluster of GPUs. Another difference is that in this thesis the covariance matrices are assumed to be diagonal while Pangborn's algorithm is developed for a general case.

The GPU accelerated expectation maximization algorithm proposed by Pangborn consists of six kernels in total. Of these six kernels, two are for expectation step while the remaining four are for maximization step. The last kernel in the maximization step of Pangborn's implementation inverts the covariance matrices of Gaussians calculated, calculates the weights of each Gaussian and computes a constant that is shared by all likelihood computations. On the other hand, in this

thesis instead of having a separate kernel, covariance matrix inversion and Gaussian model weight calculations are done in other kernels. In addition to this, it is observed that the constants are same for all Gaussian models and do not have an effect on model learning. Thus there is no need for constant calculation and hence the last kernel is completely removed.

4.4.3.1.1 Expectation Step

Before the EM algorithm is run, initial mean vectors are selected randomly from the training samples on CPU and their memory indices are copied to GPU. Initial covariance matrices are selected as identity matrix for all models and model weights are given equally. As it is mentioned, expectation step consists of two kernels.

The first kernel computes the likelihood and the log-likelihood of a training sample patch $\boldsymbol{p_i}$ belongs to a Gaussian model $\boldsymbol{M_j}$ with a mean vector $\boldsymbol{\mu_j}$, a covariance matrix $\boldsymbol{\Sigma_j}$ and weight w_j as given in (23) and (24).

$$p(\boldsymbol{p_i}|\boldsymbol{M_j}) = \exp\{-\tfrac{1}{2}(\boldsymbol{p_i}-\boldsymbol{\mu_j})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{p_i}-\boldsymbol{\mu_j})\} \tag{23}$$

$$LL(\boldsymbol{p_i}, \boldsymbol{M_j}) = -\tfrac{1}{2}(\boldsymbol{p_i}-\boldsymbol{\mu_j})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{p_i}-\boldsymbol{\mu_j}) \tag{24}$$

If the weight of the Gaussian is included the (23) and (24) become (25) and (26).

$$p(\boldsymbol{p_i}|\boldsymbol{M_j})' = \mathrm{w_j}\exp\{-\tfrac{1}{2}(\boldsymbol{p_i}-\boldsymbol{\mu_j})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{p_i}-\boldsymbol{\mu_j})\} \tag{25}$$

$$LL(\boldsymbol{p_i}, \boldsymbol{M_j})' = -\tfrac{1}{2}(\boldsymbol{p_i}-\boldsymbol{\mu_j})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{p_i}-\boldsymbol{\mu_j}) + \ln(w_j) \tag{26}$$

Since all log-likelihood computations are independent, for each computation a thread is assigned. The number of thread blocks should be a multiple of the number of Gaussian models to be learned. The thread blocks can also be configured as a two dimensional grid. Then the size of one dimension becomes the number of Gaussian models to be learned, and the other dimension can be selected according to the number of training samples.

The kernel implementation is given in Listing 13. If it is the first iteration of the EM algorithm, the kernel initializes model means, covariance matrices and weights. This implementation caches the parameters (mean vector and covariance matrix) for the Gaussian models in shared memory since they are used repeatedly for every likelihood calculation. The log-likelihood is computed rather than likelihood for the sake of numerical stability since the exponential function can overflow a 32-bit floating point number with an input as small as 90. The implementation supports only the use of diagonal covariance and since the covariance matrix is diagonal, only the diagonal values are stored.

```
function kernel_expectation_step_1(d_trbuffer, d_means, d_invCovs,
d_weights, d_probs, d_irand, d_max, isFirstIt)
        declare float s_mean, s_invCov in shared memory
        declare float log_weight
        declare int cid ← blockIdx.y
        declare int gid ← threadIdx.x + blockIdx.x * blockDim.x
        declare int offset ← blockDim.x * gridDim.x
        declare uint ntr ← d_max
        if isFirstIt = true then
                log_weight ← log(1f/gridDim.y)
        else
                log_weight ← log(d_weights[cid])
        end if
        if threadIdx.x = 0 then
                declare int i ← 0
                repeat
                        if isFirstIt = true then
                                s_mean[i] ← d_trbuffer [i*n_tr+d_irand[cid]]
                                s_invCov[i] ← 1
                        else
                                s_mean[i] ← d_means[cid * N_BINS_SQR + i]
                                s_invCov[i] ← d_invCovs[cid * N_BINS_SQR + i]
                        end if
                        i← i+1
                until i >= N_BINS_SQR
        end if
        synchronize block
        repeat
                declare float LL ← 0
                declare int i ← 0
                repeat
                        declare float data ← d_trbuffer [i*n_tr+gid]
                        LL ← LL + s_invCovs[i]*(data-s_mean[i])*(data-
s_mean[i])
                        i← i+1
                until i >= N_BINS_SQR
                d_probs[cid*n_tr+gid] ← -0.5 * LL + log_weight
                gid ← gid + offset
        until gid >= n_tr
        return
```

Listing 13 - Kernel implementation for the first step of Expectation part.

```
function kernel_expectation_step_2(d_probs, d_likelihood, d_max)
        declare float s_LL in shared memory
        declare int gid ← threadIdx.x + blockIdx.x * blockDim.x
        declare int offset ← blockDim.x * gridDim.x
        declare uint ntr ← d_max
        s_LL ← 0
        repeat
                declare float max ← max(d_probs)
                declare float denominator ← 0
                declare int c ← 0
                repeat
                        denominator ← denominator + exp(d_probs[c*n_tr+gid]-
max)
                        c ← c+1
                until c >= N_CLUSTERS
                denominator ← max + log(denominator)
                s_LL[threadIdx.x] ← s_LL[threadIdx.x] + denominator
                c ← 0
                repeat
                        d_probs[c*n_tr+gid] ← exp(d_probs[c*n_tr+gid]-
denominator)
                        c ← c+1
                until c >= N_CLUSTERS
                gid ← gid + offset
        until gid >= n_tr
        synchronize block

        declare s ← blockDim.x >> 1
        repeat
                if threadIdx.x < s then
                        s_LL[threadIdx.x] ← s_LL[threadIdx.x]+
s_LL[threadIdx.x+s]
                end if
                synchronize block
                s ← s >> 1
        until s <= 0

        if threadIdx.x = 0 then
                d_likelihood[blockIdx.x] ← s_LL[threadIdx.x]
        end if
        return
```

Listing 14 - Kernel implementation for expectation step 2.

The second kernel of expectation step converts each weighted likelihood into a fuzzy probability membership of each Gaussian model using (27) and undoes the logarithm at the end and obtains the result of (28).

$$LL(\boldsymbol{M_j}, \boldsymbol{p_i}) = \frac{LL(\boldsymbol{p_i}, \boldsymbol{M_j})'}{\ln\{\sum_{l=1}^{k} p(\boldsymbol{p_i}|\boldsymbol{M_l}) w_l\}} \tag{27}$$

$$p(\boldsymbol{M_j}|\boldsymbol{p_i}) = \frac{\mathrm{w_j} \exp\left\{-\frac{1}{2}(\boldsymbol{p_i}-\boldsymbol{\mu_j})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{p_i}-\boldsymbol{\mu_j})\right\}}{\sum_{l=1}^{k} p(\boldsymbol{p_i}|\boldsymbol{M_l}) w_l} \tag{28}$$

Since the first step computes log-likelihood rather than just likelihood, a log sum of exponentials must be used for the denominator. The membership computation is independent for each training sample, thus the number of parallel operations are the same as the number of training samples. As a result, for each training sample one thread is assigned and the number of thread blocks can be assigned according to the number of training samples. Since there is a parallel reduction, the number of threads in each thread block should be a power of two. The amount of shared memory used per block is same as the number of threads in each block. The kernel implementation for the second step is given in Listing 14.

4.4.3.1.2 Maximization Step

Maximization step of the parallel EM algorithm consists of three separate kernels each of which updates the parameters of Gaussian models according to the fuzzy membership values calculated in the expectation step.

The first kernel is a simple one that computes the size of each model by adding all fuzzy membership values as it is given in (29) for a total of N training samples. The kernel also computes the weight of each model using this sum with (30). For each Gaussian model, one thread block is assigned and since there is a parallel reduction, the number of threads in each thread block should be a power of two. The amount of shared memory used per block is the same as the number of threads in each block. The implementation is given in Listing 15.

$$N_j = \sum_{n=1}^{N} p(\boldsymbol{M_j}|\boldsymbol{p_i}) \tag{29}$$

$$w_j = \frac{N_j}{N} \tag{30}$$

The second kernel computes m^2 dimensional mean vector for each Gaussian model using (31). It is launched with a 2-dimensional grid of blocks. The size of one dimension of the grid is the number of Gaussians to be learned (k) and the size of the other dimension is the dimension of the feature space (m^2).

```
function kernel_maximization_step_n(d_probs, d_n, d_weights, d_max)
        declare float sums in shared memory
        declare int m ← blockIdx.x
        declare int offset ← blockDim.x
        declare uint ntr ← d_max
        declare float sum ← 0
        declare int i ← threadIdx.x
        repeat
                sum  ← sum + d_probs[m*n_tr+i]
                i ← i + offset
        until i >= n_tr
        sums[threadIdx.x]  ← sum
        synchronize block
        declare s ← blockDim.x >> 1
        repeat
                if threadIdx.x < s then
                        sums[threadIdx.x]  ← sums[threadIdx.x] +
sums[threadIdx.x+s]
                end if
                synchronize block
                s ← s >> 1
        until s <= 0
        if threadIdx.x = 0 then
                declare float dummy_n ← sums[threadIdx.x]
                d_n[m] ← dummy_n
                d_weights[m] ← dummy_n / n_tr
        end if
        return
```

Listing 15 - Kernel implementation for model size and weight computation.

$$\boldsymbol{\mu_j} = \frac{1}{N_j}\sum_{n=1}^{N} \boldsymbol{p_n}\, p(\boldsymbol{M_j}|\boldsymbol{p_n}) \tag{31}$$

The implementation of the kernel is very similar to the previous kernel and includes parallel reduction. Thus, the number of threads in each block should be a power of two. Again the amount of shared memory used per block is the same as the number of threads in each block. The implementation of the kernel is given in Listing 16.

```
function kernel_maximization_step_means(d_trbuffer, d_probs, d_means, d_n,
d_max)
        declare float sums in shared memory
        declare int m ← blockIdx.x
        declare int d ← blockIdx.y
        declare int offset ← blockDim.x
        declare uint ntr ← d_max
        declare float sum ← 0
        declare int i ← threadIdx.x
        repeat
                sum  ← sum+d_trbuffer[d*n_tr+i]* d_probs[m*n_tr+i]
                i ← i + offset
        until i >= n_tr
        sums[threadIdx.x]  ← sum
        synchronize block
        declare s ← blockDim.x >> 1
        repeat
                if threadIdx.x < s then
                        sums[threadIdx.x]  ← sums[threadIdx.x] +
sums[threadIdx.x+s]
                end if
                synchronize block
                s ← s >> 1
        until s <= 0
        if threadIdx.x = 0 then
                d_means[m*N_BINS_SQR+d] ← sums[threadIdx.x]/d_n[m]
        end if
        return
```

Listing 16 - Kernel implementation for mean vector computation.

The last kernel computes the covariance matrices of Gaussian models using (32) and stores them by inverting. It also is very similar to the kernel that computes means of the Gaussian models. Since the covariance matrices are assumed to be diagonal and only the diagonal values are stored; a two dimensional grid of blocks is launched. Same as the mean computation kernel, the size of the one dimension of the grid is the number of models to be learned (k) and the other is the dimension of the feature space (m^2). Due to the parallel reduction, the number of threads in each block should be a power of two. The amount of shared memory per block is again the same as the number of threads in each thread block. The implementation is given in Listing 17.

$$\boldsymbol{\Sigma_j} = \frac{1}{N_j}\sum_{n=1}^{N}(\boldsymbol{p_n} - \boldsymbol{\mu_j})(\boldsymbol{p_n} - \boldsymbol{\mu_j})^T p(\boldsymbol{M_j}|\boldsymbol{p_n}) \quad (32)$$

Expectation and maximization steps are called iteratively until the likelihood change drops below some predetermined threshold or the number of iterations is larger than the max iteration number determined.

```
function kernel_maximization_step_covariance(d_trbuffer, d_probs, d_means,
d_invCovs, d_n, d_max)
        declare float sums in shared memory
        declare int m ← blockIdx.x
        declare int d ← blockIdx.y
        declare int offset ← blockDim.x
        declare uint ntr ← d_max
        declare float sum ← 0
        declare int i ← threadIdx.x
        repeat
                declare float mean ← d_means[m*N_BINS_SQR+d]
                declare float tr ← d_trbuffer[d*n_tr+i]
                sum  ← sum+(tr-mean)*(tr-mean)*d_probs[m*n_tr+i]
                i ← i + offset
        until i >= n_tr
        sums[threadIdx.x]  ← sum
        synchronize block
        declare s ← blockDim.x >> 1
        repeat
                if threadIdx.x < s then
                        sums[threadIdx.x]  ← sums[threadIdx.x] +
sums[threadIdx.x+s]
                end if
                synchronize block
                s ← s >> 1
        until s <= 0
        if threadIdx.x = 0 then
                d_invCovs[m*N_BINS_SQR+d] ← d_n[m]/sums[threadIdx.x]
        end if
        return
```

Listing 17 - Kernel implementation for diagonal covariance matrix computation.

4.4.3.2 Model Updating

Same as the CPU implementation, after the first frame in each new frame, the old models are updated using freshly gathered training samples. Model updating part is divided into two kernels. The kernel implementation for the first kernel is given in Listing 18.

```
function kernel_update_models_step1(d_trbuffer, d_trinds, d_means.
d_weights, d_invCovs, n_models, threshold, ntr)
        declare float sum_mean, sum_invCov, sum_weight in shared memory
        declare int gid ← blockIdx.x * blockDim.x + threadIdx.x
        declare int tid ← threadIdx.x
        declare int offset ← blockDim.x * gridDim.x
        declare int n_tr ← ntr
        declare int n_tr_removed ← 0
        initialize sum_mean, sum_invCov, sum_weight to 0
        synchronize block
        repeat
                declare float wsample ← 1f/n_tr
                declare float mindist ← INF
                declare uint minind ← 0
                mindist, minind ← find closest model to d_trbuffer[gid]
                if mindist < threshold then
                        d_trinds[gid] ← 0
                        declare int i ← 0
                        repeat
                                declare int ind ← minind * N_BINS_SQR + i
                                declare float sample ← d_trbuffer[i*n_tr+gid]
                                declare float mean ← d_means[ind]
                                atomicAdd(sum_mean[ind], wsample * sample)
                                atomicAdd(sum_invCov[ind], wsample * (sample-
mean)^2)
                                i ← i + 1
                        until i >= N_BINS_SQR
                        atomicAdd(sum_weight[minind], weight_sample)
                        n_tr_removed ← n_tr_removed + 1
                end if
                gid ← gid + offset
        until gid>= n_tr
```

```
        synchronize block
        atomicSub(ntr, n_tr_removed)
        tid ← threadIdx.x
        repeat
                if blockIdx.x = 0 then
                        declare float weight ← d_weights[tid/N_BINS_SQR]
                        declare float invC ← d_invCovs[tid]
                        d_means[tid] ← d_means[tid] * weight
                        d_invCovs[tid] ← weight / invC
                end if
                atomicAdd(d_means[tid],sum_mean[tid])
                atomicAdd(d_invCovs[tid],sum_invCov[tid])
                atomicAdd(d_weights[tid/N_BINS_SQR],
sum_weight[tid/N_BINS_SQR])
                tid ← tid + blockDim.x
        until tid >= N_BINS_SQR * n_models and ntr = n_tr
        return
```

Listing 18 - Kernel for the first step of model updating.

Initially, the first kernel finds the minimum Mahalanobis distance from each training sample to the models and the closest model to each sample using (3). After that for the samples that have a minimum distance smaller than the update threshold (T_m), the terms, which are the sample related terms in (4), (5) and (6), given in (33), (34) and (35) are calculated and added to the related mean, covariance and weight dimension in the shared memory. Those training samples which are used to update the available models are marked as zero in the training buffer.

$$w_i \boldsymbol{x_i} \tag{33}$$

$$w_i \left((\boldsymbol{x_i} - \boldsymbol{\mu_j})(\boldsymbol{x_i} - \boldsymbol{\mu_j})^T \right) \tag{34}$$

$$w_i \tag{35}$$

After these partial sums are calculated, the terms in the numerator of (4), (5) and (6), which can be given in (36), (37) and (38), are calculated at the end of the kernel using (33), (34) and (35). When the kernel is launched, one thread is assigned to each training sample and all training samples are divided into blocks. Since shared memory is used to hold partial sums in each block, the size of the shared memory used for each block is the same as the memory used by the model parameters.

$$(w_j \boldsymbol{\mu_j} + w_i \boldsymbol{x_i}) \tag{36}$$

$$(w_j \boldsymbol{\Sigma_j} + w_i \left((\boldsymbol{x_i} - \boldsymbol{\mu_j})(\boldsymbol{x_i} - \boldsymbol{\mu_j})^T \right) \tag{37}$$

$$(w_j + w_i) \tag{38}$$

```
function kernel_update_models_step1(d_means. d_weights, d_invCovs,
n_models)
        declare int gid ← blockIdx.x * blockDim.x + threadIdx.x
        declare int offset ← blockDim.x * gridDim.x
        repeat
                declare float weight ← d_weights[gid/N_BINS_SQR]
                declare float invC ← d_invCovs[gid]
                d_means[gid] ← d_means[gid] / weight
                d_invCovs[gid] ← weight / invC
                gid ← gid + offset
        until gid >= N_BINS_SQR * n_models
        return
```

Listing 19 - Kernel implementation for the second step of model updating scheme.

For the division to the total weight and hence the total application of (4), and (5) for means and covariances, another kernel is run. Since the total number of models is relatively small, this computation can also be done on CPU. Normalization of weights using (6) is done on CPU by taking weight values from GPU memory to CPU memory since the number of data to be processed is too small. In the second kernel one thread is assigned to each dimension of weight or covariance, which results in a maximum Km^2 number of threads. If Km^2 is sufficiently large, the kernel can be called with more than one block of threads. The kernel implementation for this step is given in Listing 19.

While some training samples are used for updating, some others are not. If there are no training samples that are not used for updating, the algorithm moves to the classification part. However, if there are some training samples which are not used for updating, new k models will be learned from these training samples. Those remaining training samples are compacted to another training buffer. In order to find the memory indices of these remaining training samples, first the compact scan kernel algorithm given in Listing 11 used before for the filtering of training samples in the automatic training sample generation section is called. After that another kernel

which is very similar to the kernel given in Listing 12 is called to insert the specified training samples to another array ($d\_trbuffer\_remaining$). This kernel is given in Listing 20.

```
function kernel_insert_training_samples(d_trlabels, d_trcompact, d_trbuffer,
d_trbuffer_remaining, d_max, d_max_old, n_patches)
        declare uint gid ← blockIdx.x * blockDim.x + threadIdx.x
        declare uint offset ← blockDim.x * gridDim.x
        declare uint n_tr ← d_max
        declare uint n_tr_old ← d_max_old
        repeat
                if d_trlabels[gid] != 0 then
                        declare uint id ← d_trcompact[gid]
                        declare int i ← 0
                        repeat
                                d_trbuffer_remaining[i*n_tr+id] ← d_trbuffer
[i*n_tr_old+gid]
                                i ← i + 1
                        until i >= N_BINS_SQR
                end if
                gid ← gid + offset
        until gid >= n_patches
        return
```

Listing 20 - Kernel implementation for inserting the remaining training samples.

After those remaining samples are gathered in a separate array, GPU-accelerated expectation maximization algorithm explained in the previous section is run to learn new k models. Those new k Gaussian models are then put into model library.

However, the number of Gaussian models in the library can be at its maximum (K), that is, the model library can be full. Thus, on CPU, it is first checked if model library is full or not. Then, if model library is full, by checking the weights of the old models, the indices of the models with the smallest weights are found. If model library is not full, these indices are assigned as the empty indices in the library. After the indices are determined to insert new models to the model library another kernel is called. Note that, since the number of parameters to be inserted is small, this part can be done on CPU and the performance might be better. However, in that case some part of GPU memory should be copied to CPU memory and after the operation, this memory

should be copied back to the GPU memory. This memory copies create an overhead and might degrade the performance. In this study it is implemented on GPU.

```
function kernel_replace_models (d_means. d_weights, d_invCovs,
d_means_new. d_weights_new, d_invCovs_new,
d_ind_clusters_min_weights)
        declare uint m ← blockIdx.x
        declare uint d ← threadIdx.x
        declare uint m_replaced ← d_ind_clusters_min_weights[m]
        declare uint indr ← m_replaced * N_BINS_SQR + d
        declare uint ind ← m * N_BINS_SQR + d
        d_means[indr] ← d_means_new[ind]
        d_invCovs[indr] ← d_invCovs_new[ind]
        if threadIdx.x = 0 then
                d_weights[m_replaced] ← d_weights_new[m]
        end if
        return
```

Listing 21 - Kernel implementation for placing the new learned models into model library.

```
function kernel_classification_raw(d_hists, d_classified, d_trlabels, d_means,
d_invCovs, size, n_models, threshold)
        declare uint gid ← blockIdx.x * blockDim.x + threadIdx.x
        declare uint offset ← blockDim.x * gridDim.x
        repeat
                declare float mindist ← find minimum mahalanobis distance
between models and the patch gid
                if mindist < threshold then
                        d_classified[gid] ← 1
                else
                        d_classified[gid] ← 0
                end if
                gid ← gid + offset
        until gid >= size
        return
```

Listing 22 - Kernel implementation of the first step of classification.

The kernel implementation for new model placing is given in Listing 21. In the kernel call, for each model to be placed into the model library, one thread block is called. The number of threads in each block is the size of the dimension of the feature space, namely, m^2. After this kernel call, weights of the Gaussians are copied to CPU memory and normalized such that the summation of all model weights is 1. This is required since with the newly learned models, the summation of weights is not 1 anymore. Since number of models is relatively small and weight parameter is only a scalar, the number of data to be processed is small and hence, this computation is done on CPU.

4.4.4 Classification

After model learning and updating, the last part is the classification of whole image using the Gaussian models in the model library. In CPU implementation, classification is done through region growing from the training samples obtained from the frame. Since region growing is inefficient to run on a highly parallel hardware, another approach that yields the same result and appropriate for parallelization is used. In this respect, the minimum Mahalanobis distance between each patch's color feature vector in the image and the models in the model library is found first. If this distance is smaller than classification threshold (T_c), this patch is marked as road and else it is marked as non-road in a binary image ($d\_classified$). This binary image holds the road/non-road value of each image patch. If the patch is marked as road, its value is 1, else its value is 0. After that, connected components labeling is applied to the binary image to find out the connected regions in the binary image. At last, the connected component which includes one of the training samples obtained in that frame ($d\_roadlabel$) is filtered and only that connected component is labeled as road.

As it is mentioned, the first kernel evaluates the Mahalanobis distance between a patch and all models in the model library, and finds the minimum of them. If this minimum distance is smaller than classification threshold (T_c), it is marked as road on a binary image, else marked as non-road. Since Mahalanobis distance is found as the most appropriate distance metric for the classification in the previous chapter, it is implemented and used in the classification. The whole patch image is divided into tiles and each tile is assigned to a thread block, while a thread is assigned to a patch. The number of threads to run (N_t) and number of blocks to run (N_b) should be selected such that the multiplication of these two ($N_b x N_t$) should be a divider of the total number of patches. The implementation for this simple kernel is given in Listing 22.

After this thresholding, classified image ($d\_classified$) is fed to the connected components labeling algorithm given in Listing 7 and Listing 8. The algorithm in Listing 7 is run completely same as in the previous "Connected Components Labeling" section. This time the input to the initialization kernel is the found classified binary image ($d\_classified$). However, the second kernel of connected components labeling algorithm given in Listing 8 has minor differences. Instead of the size calculation of each connected component, this time the local neighbor propagation kernel returns the label of the connected component having the training samples. To be able to do this, the location of one of the training samples found in the "automatic generation

of training samples" part is fed to the kernel (d_maxgid). The modified kernel implementation is given in Listing 23. Same as before, the algorithm runs iteratively until there is no change in the connected component labels.

In the last kernel, by using the label of the connected component including training samples found in the connected components labeling algorithm ($d\_roadlabel$), all classified image is filtered. Thus, only the patches of the connected component with the label found in the previous kernel remain as road and all other patches marked as non-road. Again the whole patch image is divided into tiles and for each tile a thread block is assigned. Each thread is assigned to a patch. The number of threads to run (N_t) and number of blocks to run (N_b) should be selected such that the multiplication of these two ($N_b x N_t$) should be a divider of the total number of patches. The kernel implementation for filtering is given in Listing 24.

After the filtering operation, the classified road image is obtained on GPU memory. Then, the classified binary image ($d\_classified$) is copied from the GPU memory the CPU memory.

4.5 Experiments and Results

In this section the running speed of both the CPU and GPU algorithms are evaluated and compared to each other. For a configuration selected as reference, each of the algorithm's parameters is changed one at a time to see the effect of the parameters on the performance and the parameters that yield the best performance are tried to find. The reference algorithm parameters selected are given in Table 3. As the effect of parameter (N) is found insignificant in the previous chapter, it is not included. On the other hand, since the GPU has special parameters that should be selected, number of threads to run (N_t) and number of blocks to run (N_b) are introduced as new parameters to be optimized.

Table 7 - Reference values for algorithm parameters to evaluate the running speed effect of each parameter.

Parameter	$T_r\ (m^2)$	$n\ (pixels)$	$m\ (bins)$	k_0	k	K	T_m	T_c	N_b	N_t
Value	0.00001	5	8	1	1	10	1	5	12	256

```
function kernel_ccl_mesh_B_improved_classify(d_classified, d_roadlabel,
d_max_gid, m_d, widthgrid, heightgrid)
        declare int nid in local memory
        declare int Ll, ml in shared memory
        declare int id ← (threadIdx.y + blockIdx.y * blockDim.y) * widthgrid +
threadIdx.x + blockIdx.x * blockDim.x
        declare int idl ← threadIdx.y * blockDim.x + threadIdx.x
        declare int idn ← 0
        declare uint label ← texfetch(d_trlabels, id)
        ml ← 1
        if label != 0 then
                if (id - widthgrid) >= 0 and label * texfetch (d_trlabels, id-
widthgrid)
                        nid[idn] ← texfetch (d_trlabels, id-widthgrid)
                        idn ← idn + 1
                end if
                if (id + widthgrid) < widthgrid * heightgrid and label *
tex1Dfetch(d_trlabels, id+widthgrid)
                        nid[idn] ← texfetch (d_trlabels, id+widthgrid)
                        idn ← idn + 1
                end if
                if ((id - 1) % widthgrid) != widthgrid - 1 and label *
        tex1Dfetch(d_trlabels, id-1)
                        nid[idn] ← texfetch (d_trlabels, id-1)
                        idn ← idn + 1
                end if
                if ((id + 1) % widthgrid) != 0 and label * texfetch (d_trlabels,
        id+1)
                        nid[idn] ← texfetch (d_trlabels, id+1)
                        idn ← idn + 1
                end if
                declare int i ← 0
                repeat
                        if nid[i] < label then
                                label ← nid[i]
                                md ← 1
                        end if
                i ← i + 1
                until i >= idn
```

```
    idn ← 0
    if (idl–blockDim.x)>=0 and label* texfetch (d_trlabels, id-widthgrid)
            nid[idn] ← idl - blockDim.x
            idn ← idn + 1
    end if
    if (idl+blockDim.x)<blockDim.x*blockDim.y and label*texfetch
(d_trlabels, id+widthgrid)
            nid[idn] ← idl + blockDim.x
            idn ← idn + 1
    end if
    if ((idl - 1) % blockDim.x) != blockDim.x - 1 and label * texfetch
(d_trlabels, id-1)
            nid[idn] ← idl - 1
            idn ← idn + 1
    end if
    if ((idl+1)%blockDim.x) != 0 and label * texfetch (d_trlabels, id+1)
            nid[idn] ← idl + 1
            idn ← idn + 1
    end if
    repeat
            Ll[idl] ← label
            synchronize block
            ml ← 0
            i ← 0
            repeat
                    if Ll[nid[i]] < label then
                            label ← Ll[nid[i]]
                            ml ← 1
                    end if
            synchronize block
            i ← i + 1
            until i >= idn
            d_trlabels[id] ← label
            if id = d_maxgid then
                    d_roadlabel ← label
            end if
        until ml != true
end if
return
```

Listing 23 - Kernel implementation for modified local neighbour propagation kernel. Modified from the kernel given in Listing 8.

```
function kernel_filter(d_trlabels, d_roadlabel, n_patches)
        declare uint gid ← blockIdx.x * blockDim.x + threadIdx.x
        declare uint roadlabel ← d_roadlabel
        repeat
                if d_trlabels[gid] = roadlabel then
                        d_trlabels[gid] ← 1
                else
                        d_trlabels[gid] ← 0
                end if
                gid ← gid + gridDim.x * blockDim.x
        until gid >= n_patches
        return
```

Listing 24 - Kernel implementation for the filtering of connected components.

The CPU algorithm is run on a Intel Core i7 3970K processor while the GPU algorithm is run on an NVidia Geforce GTX 650 graphics card. GTX 650 is a low-end graphics card with compute capability 3.0 and owns 384 CUDA cores. According to its architecture (Kepler architecture) this GPU has two streaming multiprocessors. For evaluation, the running time of the algorithm on CPU and GPU is compared. For GPU algorithm, the running time includes the time passing while the data is transferring from CPU to GPU memory and from GPU to CPU memory. Besides running times, the speed-ups gained using GPU algorithm over CPU algorithm are also given as performance metrics.

The first experiment is done to find the optimal value of (N_t) the number of threads to run. In Table 8, the the running time of the GPU algorithm with different number of threads (N_t) run are given. The mean, minimum and maximum of the running times of 360 frames are given as the performance indicators. According to these results, running 256 or 512 threads yields approximately same performances while running smaller number of threads degrades the performance significantly.

The next experiment is done on the number of blocks to run (N_b). Again, the mean, minimum and maximum of the running times of the 360 frames are given as performance indicators. As Table 9 suggests, running blocks more than 8 yields the best performance while 8 and smaller numbers degrades the performance since the resources of the graphics card are not utilized fully.

Table 8 - The effect of number of threads to run on the algorithm's running speed performance. Mean, min and max of 360 frames are given.

Number of Threads (N_t)	**32**	**64**	**128**	**256**	**512**
Mean (ms)	22.76	16.01	11.94	10.60	10.75
Min (ms)	14.09	9.66	7.98	8.17	8.84
Max (ms)	29.03	21.81	19.50	15.36	15.13

Table 9 - The effect of number of blocks to run on the algorithm's running speed performance. Mean, min and max of 360 frames are given.

Number of Blocks (N_b)	**6**	**8**	**12**	**16**	**24**	**48**
Mean (ms)	11.97	11.04	10.63	10.38	10.51	10.19
Min (ms)	8.68	8.60	8.47	7.71	8.62	7.70
Max (ms)	16.76	15.18	14.91	13.92	14.44	14.43

When the contribution of all algorithm parameters to algorithm complexity is considered, the thresholds T_r, T_m and T_c do not have a direct effect on the running time. On the other hand, the situation is different for other parameters. For an image of specific size, patch size (n) determines how many patch to be processed. As n decreases, there will be more patch to be processed. As a result the running time of the algorithm becomes larger. Table 10 shows mean, minimum and maximum running times of 360 frames both for CPU and GPU with different patch size values. As it can be seen from the table, GPU algorithm is affected lesser due to its parallel nature and can work at least 50 Hz while CPU algorithm works in 1-40 Hz depending on the configuration. Even with the lightest configuration the speed of CPU algorithm drops up to 6 Hz, which cannot be considered as real-time.

Table 11 shows the speedup gain of the GPU algorithm over CPU algorithm for different patch sizes. According to this table, the GPU algorithm can work 2.7 times to 62 times faster than the CPU version. Since small n means more data to be processed, the parallel work that should be done by GPU with its resources is more and hence the speedup gain becomes larger.

Table 10 - CPU and GPU algorithm running times with changing patch size. Mean, min and max of 360 frames are given.

Run Time	$n = 4$		$n = 5$		$n = 8$	
	CPU	**GTX 650**	**CPU**	**GTX 650**	**CPU**	**GTX 650**
Mean (ms)	434.99	11.27	269.59	10.38	89.32	7.78
Min (ms)	77.96	7.60	54.98	7.71	25.76	6.07
Max (ms)	669.67	17.04	443.19	13.92	146.74	11.24

Table 11 - GPU algorithm speedup over CPU algorithm with changing patch sizes. Mean, min and max of 360 frames are given.

Speedup	$n = 4$	$n = 5$	$n = 8$
Mean (x)	38.81	26.05	11.59
Min (x)	5.74	4.83	2.72
Max (x)	61.94	41.10	20.43

Another parameter having a large effect on the speed of the algorithm is number of histogram bins (m). Since the feature vectors used in the algorithm are actually H-S joint histograms of image patches, as (m) increases, the dimension of feature space increases with its square. The dimension of the feature space merely affects the speed of all types of matrix operations and also the speed of model learning. Table 12 shows mean, minimum and maximum running times of 360 frames both for CPU and GPU with different number of histogram bins. As it can be seen from the table, GPU algorithm can work at least 50 Hz while CPU algorithm works in 1-20 Hz depending on the configuration. Even with the lightest configuration the speed of CPU algorithm drops up to 4 Hz, which cannot be considered as real-time.

Table 13 shows the speedup gain of the GPU algorithm over CPU algorithm for different number of histogram bins. According to this table, the GPU algorithm can work 5.4 times to 66 times faster than the CPU version. Since high m means more data to be processed, the GPU resources can be fully utilized and hence as m grows, the speedup becomes larger.

Table 12 - CPU and GPU algorithm running times with changing number of histogram bins. Mean, min and max of 360 frames are given.

Run Time	$m = 4$		$m = 6$		$m = 8$		$m = 10$	
	CPU	**GTX 650**	**CPU**	**GTX 650**	**CPU**	**GTX 650**	**CPU**	**GTX 650**
Mean (ms)	188.30	8.95	193.34	9.32	269.59	10.19	415.23	11.58
Min (ms)	50.62	6.81	45.74	6.96	54.98	7.70	58.32	8.01
Max (ms)	247.06	12.72	282.69	14.06	443.19	14.43	746.48	15.94

Table 13 – GPU algorithm speedup over CPU algorithm with changing number of histogram bins. Mean, min and max of 360 frames are given.

Speedup	$m = 4$	$m = 6$	$m = 8$	$m = 10$
Mean (x)	21.32	20.88	26.50	35.80
Min (x)	6.00	5.62	5.40	5.37
Max (x)	31.86	32.15	43.62	65.91

The number of models learned in the first frame (k_0) and the number of models learned in each new cycle (k) can only affect the speed of the learning. Usually the values of these parameters are so small that their effect becomes insignificant compared to the other parameters and hence they are not investigated.

The last parameter having a considerable effect on algorithm running speed is the maximum size of the model library (K). This parameter determines how many Gaussian models to be held in the memory of the system. As the number of models in the model library increases, there will be more and more comparisons done in both model updating and classification steps. As a result as this parameter becomes larger the algorithm speed decreases. Table 14 shows mean, minimum and maximum running times of 360 frames both for CPU and GPU with different model sizes. As it can be seen from the table, GPU algorithm can work at least 50 Hz while CPU algorithm works in 0.5-20 Hz depending on the configuration. Even with the lightest configuration the speed of CPU algorithm drops up to 4 Hz, which cannot be considered as real-time. Since model library is initially empty, minimum running time of all configurations for both CPU and GPU versions are approximately the same. However, after the model library is full CPU algorithm becomes very small compared to the GPU algorithm.

Table 15 shows the speedup gain of the GPU algorithm over CPU algorithm for increasing model library sizes. According to this table, the GPU algorithm can work 3.5 times to 110 times faster than the CPU version. As K becomes larger the parallelization becomes more advantageous.

Table 14 - CPU and GPU algorithm running times with changing model library size. Mean, min and max of 360 frames are given.

	Mean (ms)		Min (ms)		Max (ms)	
	CPU	GTX 650	CPU	GTX 650	CPU	GTX 650
$K = 5$	166.60	9.85	53.28	7.94	255.01	14.60
$K = 10$	345.01	10.63	50.26	8.47	652.78	14.91
$K = 15$	379.94	11.48	53.67	8.20	615.04	15.66
$K = 20$	460.13	12.23	46.43	7.98	719.27	18.46
$K = 25$	592.01	12.85	54.81	8.49	937.25	16.97
$K = 30$	709.00	13.50	56.15	8.33	1122.82	17.71
$K = 35$	793.30	14.20	48.97	8.00	1463.77	19.06
$K = 40$	922.14	14.94	54.87	8.11	1525.85	20.90
$K = 45$	1031.04	15.61	55.18	7.87	1676.24	19.69
$K = 50$	1148.04	16.29	54.46	8.01	1947.75	20.84

4.6 Conclusion

In this chapter, the adaptive self-learning unstructured road detection algorithm presented in the previous chapter is parallelized and implemented on a CUDA accelerated GPU using CUDA C. The experiments are done to show the effect of the algorithm parameters on the algorithm running speed. The results show that even on a low-end graphics card, the algorithm can work at 50 Hz in the worst case. Considering an autonomous mobile field robot, this performance is more than enough.

Table 15 - GPU algorithm speedup over CPU algorithm with changing model library size. Mean, min and max of 360 frames are given.

Speedup	Mean (x)	Min (x)	Max (x)
$K = 5$	17.01	5.22	25.14
$K = 10$	32.58	4.38	69.94
$K = 15$	33.16	4.60	54.16
$K = 20$	37.58	4.00	57.84
$K = 25$	45.85	4.43	70.25
$K = 30$	52.10	4.58	85.21
$K = 35$	55.31	3.72	97.86
$K = 40$	61.00	3.88	94.84
$K = 45$	65.09	3.69	105.45
$K = 50$	69.30	3.52	110.18

CHAPTER 5

CONCLUSION & FUTURE WORK

5.1 Conclusion

In this thesis study an adaptive self-learning unstructured road detection algorithm which is suitable using on autonomous or semi-autonomous mobile field robots is proposed. The stereo data obtained using a stereo camera is used as the only information input to the robot. Processing both geometrical and color information streamed by the stereo camera, the robot can be able to find the traversable places without being affected light conditions.

In this respect, a literature review on the unstructured road detection techniques is done to learn the state of the art. Examining the drawbacks of the techniques in the literature, a new technique is developed and implemented. The implemented solution is tested against a dataset captured with a stereo camera along the pathways of Yalincak village in METU Campus. Using the results of the experiments done, the algorithm parameters are optimized to have satisfactory classification results.

Since the algorithm works slowly, in order to increase its processing speed to real-time the developed algorithm redesigned as parallel and implemented on a CUDA enabled NVidia GPU. With the parallelization of the algorithm the working speed of the algorithm becomes 50 Hz in the worst case and hence real-time considerations are satisfied.

5.2 Future Work

In this study the algorithm cannot be tested against very different fields. As the most important future work, the algorithm will be tested in different environment types and especially in shadowy regions. Since the algorithm is developed to work on a mobile robot, it will be deployed on an unmanned ground vehicle and tested on it.

As mentioned in the third chapter, roughness threshold (T_r) can be changed dynamically from frame to frame for more accurate road detection. Low saturation channel values of HSV color space might yield the failure of the algorithm; in the future this will be prevented. This prevention might also increase the classification accuracy of the road detection algorithm.

Finally, using not only color features on learning but color features with geometrical cues will increase the classification accuracy with a decreased false positive rate.

REFERENCES

[1] Industrial Robot: Welding (Spot Welding Robot). Image. Last accessed in February, 2011 [Online]. http://www.vp-robotics.com/robot.html

[2] Handout photo courtesy of the Nevada Department of Motor Vehicles shows a Google self-driven car. Image. Last accessed in May, 2012 [Online]. http://techland.time.com/2012/05/08/googles-driverless-cars-now-officially-licensed-in-nevada/

[3] Samsung Navibot © Samsung. Image. Last accessed in , 2012 [Online]. http://www.allonrobots.com/robot-vacuum.html

[4] D. Flood. Dennis Flood Photography. Image. Last accessed in , 2012 [Online]. http://www.dennisflood.com

[5] World Downhill Skate Championships. Image. Last accessed in September, 2005 [Online]. http://locate.irational.org/sk8map/world_downhill_skate_championships/

[6] Moonbattery. Last accessed in , 2012 [Online]. http://www.moonbattery.com/dirt-road.jpg

[7] Image. Last accessed in , 2012 [Online]. http://mw2.google.com/mw-panoramio/photos/medium/2303173.jpg

[8] Jian Wang, Zhong Ji, and Yu-Ting Su, "Unstructured Road Detection Using Hybrid Features," in *Proceedings of Machine Learning and Cybernetics, 2009 International Conference on*, 2009, pp. 482-486.

[9] M. Ekinci, F. Gibbs, and B. T. Thomas, "Knowledge-Based Navigation for Autonomous Road Vehicles," *Turkish J. of Elec. Eng. & Comp. Sci*, vol. 8, pp. 1-29, 2000.

[10] P. Jansen, W. Van Der Mark, J. C. Van Den Heuvel, and F. C. A. Groen, "Colour Based Off-Road Environment and Terrain Type Classification," in *Proceedings of Intelligent Transportations Systems, 2005 IEEE*, 2005, pp. 216-221.

[11] C. Rasmussen, "Combining Laser Range, Color, and Texture Cues for Autonomous Road Following ," in *Proceedings of Robotics and Automation, 2002 IEEE International Conference on*, 2002, pp. 4320-4325.

[12] C. Rasmussen, "Grouping Dominant Orientations for Ill-Structured Road Following," in *Proceedings of Computer Vision and Pattern Recognition, 2004 IEEE Computer Society Conference on*, 2004, pp. 470-477.

[13] C. Rasmussen, "Texture-Based Vanishing Point Voting for Road Shape Estimation," in *Proceedings of British Machine Vision Conference*, 2004.

[14] A. M. Zhang and L. Kleeman, "A Panoramic Color Vision System for Following Ill-Structured Roads," , Auckland, 2006.

[15] Hong Cheng, Nanning Zheng, and Tie Liu, "An Approach of Road Recognition Based Mean Shift and Feature Clustering," in *Proceedings of Signal Processing, 2002 6th International Conference on*, 2002, pp. 1059-1062.

[16] Minghao Hu, Wenjie Yang, Mingwu Ren, and Jingyu Yang, "A Vision Based Road Detection Algorithm," in *Proceedings of Robotics, Automation and Mechatronics, 2004 IEEE Conference on*, 2004, pp. 846-850.

[17] Jingang Huang, Bin Kong, Bichun Li, and Fei Zheng, "A New Method of Unstructured Road Detection Based on HSV Color Space and Road Features," in *Proceedings of Information Acquisition, 2007 International Conference on*, 2007, pp. 596-601.

[18] Qingji Gao, Qijun Luo, and Sun Moli, "Rough Set based Unstructured Road Detection through Feature Learning," in *Proceedings of Automation and Logistics, 2007 International Conference on*, 2007, pp. 101-106.

[19] Y. Alon, A. Ferencz, and A. Shashua, "Off-Road Path Following Using Region Classification and Geometric Projection Constraints," in *Proceedings of Computer Vision and Pattern Recognition, 2006 IEEE Computer Society COnference on*, 2006, pp. 689-696.

[20] Ceryen Tan, Tsai Hong, Tommy Chang, and Michael Shneier, "Color Model-Based Real-Time Learning for Road Following," in *Proceedings of IEEE Intelligent Transportation Systems Conference*, Toronto, 2006.

[21] L. Zheng, D. Bin, and H. Hangen, "A Novel Fast Segmentation Method of Unstructured Roads," in *Proceedings of Vehicular Electronics and Safety, 2006 IEEE International Conference on*, 2006, pp. 53-56.

[22] M. Foedisch and A. Takeuchi, "Adaptive Real-Time Road Detection using Neural Networks," in *Proceedings of The 7th International IEEE COnference on Intelligent Transportation Systems*, 2004, pp. 167-172.

[23] J. Sun et al., "Learning from Examples in Unstructured, Outdoor Environments," *Journal of Field Robotics*, vol. 23, no. 11-12, pp. 1019-1036, 2006.

[24] David Lieb, Andrew Lookingbill, and Sebastian Thrun, "Adaptive Road Following using Self-Supervised Learning and Reverse Optical Flow," in *Proceedings of Robotics Science and Systems*, Cambridge, 2005.

[25] Wei Li, Jiang Xiaojia, and Wang Yangxing, "Road Recognition for Vision Navigation of an Autonomous Vehicle by Fuzzy Reasoning," *Fuzzy Sets and Systems*, vol. 93, no. 3, pp. 275-280, 1998.

[26] J. Poppinga, A. Birk, and K. Pathak, "Hough Based Terrain Classification for Realtime Detection of Drivable Ground," *Journal of Field Robotics*, vol. 25, no. 1-2, pp. 67-88, 2008.

[27] W. Wei and S. Gong, "Research on Unstructured Road Detection Algorithm Based on the Machine Vision," in *Proceedings of Asia-Pacific Conference on Information Processing*, 2009, pp. 112-115.

[28] H. Dahlkamp, A. Kaehler, D. Stavens, S. Thrun, and G. Bradski, "Self-supervised Monocular Road Detection in Desert Terrain," in *Proceedings of Robotics: Science and Systems Conference*, Philadelphia, 2006.

[29] S. Thrun, M. Montemerlo, and A. Aron, "Probabilistic Terrain Analysis for High-Speed Desert Driving," in *Proceedings of Robotics: Science and Systems Conference*, Philadelphia, 2006.

[30] A. V. Nefian and G. R. Bradski, "Detection of Drivable Corridors for Off-Road Autonomous Navigation," in *Proceedings of Image Processing, 2006 IEEE International Conference on*, 2006, pp. 3025-3028.

[31] Boris Sofman, Ellie Lin, J. Andrew Bagnell, Nicolas Vandapel, and Anthony Stentz, "Improving Robot Navigation Through Self-Supervised Online Learning," in *Proceedings of Robotics Science and Systems*, Philadelphia, 2006.

[32] Masataka Suzuki, Teppei Saitoh, Eisuke Terada, and Yoji Kuroda, "Near-to-Far Self-Supervised Road Estimation for Complicated Environments," in *Proceedings of 5th IFAC Symposium on Mechatronic Systems*, Cambridge, 2010.

[33] Raia Hadsell et al., "Learning Long-Range Vision for Autonomous Off-Road Driving," *Journal of Field Robotics*, vol. 26, no. 2, pp. 120-144, 2009.

[34] Raia Hadsell et al., "Deep Belief Net Learning in a Long-Range Vision System for Autonomous Off-Road Driving," in *Proceedings of Proc. Intelligent Robots and Systems*, Nice, 2008.

[35] Raia Hadsell et al., "Online Learning for Offroad Robots: Using Spatial Label Propagation to Learn Long-Range Traversability," in *Proceedings of Proc. of robotics: science and systems.*, Atlanta, 2007.

[36] Raia Hadsell et al., "A Multi-Range Vision Strategy for Autonomous Offroad Navigation," in *Proceedings of Proc. Robotics and Applications*, Wuerzburg, 2007.

[37] Ayse Naz Erkan et al., "Adaptive Long Range Vision in Unstructured Terrain," in *Proceedings of Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Diego, 2007.

[38] Marco Scoffier et al., "Fully Adaptive Visual Navigation For Autonomous Vehicles," in *Proceedings of In proceedings: 27th Army Science Conference*, Orlando, 2010.

[39] Larry Matthies et al., "Learning for Autonomous Navigation: Extrapolating from Underfoot to the Far Field," *Journal of Machine Learning Research 1*, vol. 1, no. 48, pp. 1-10, 2005.

[40] Michael J. Procopio, Jane Mulligan, and Greg Grudic, "Learning in Dynamic Environments with Ensemble Selection for Autonomous Outdoor Robot Navigation," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nice, 2008.

[41] Michael J. Procopio, Jane Mulligan, and Greg Grudic, "Long-Term Learning Using Multiple Models For Outdoor Autonomous Robot Navigation," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Diego, 2007.

[42] Michael J. Procopio, Jane Mulligan, and Greg Grudic, "Learning Terrain Segmentation with Classifier Ensembles for Autonomous Robot Navigation in Unstructured Environments," *Journal of Field Robotics*, vol. 2, no. 26, pp. 145-175, 2009.

[43] Michael J. Procopio, Jane Mulligan, and Greg Grudic, "Coping with Imbalanced Training Data for Improved Terrain Prediction in Autonomous Outdoor Robot Navigation," in *Proceedings of IEEE International Conference on Robotics and Automation*, Anchorage, 2010.

[44] Michael J. Procopio, W. Philip Kegelmeyer, Greg Grudic, and Jane Mulligan, "Terrain Segmentation with On-Line Mixtures of Experts for Autonomous Robot Navigation," in *Proceedings of Proceedings of the 8th International Workshop on Multiple Classifier*

Systems, Reykjavik, 2009.

[45] Greg Grudic and Jane Mulligan, "Outdoor Path Labeling Using Polynomial Mahalanobis Distance," in *Proceedings of Robotics Science and Systems*, Philadelphia, 2006.

[46] Masataka Suzuki, Eisuke Terada, Teppei Saitoh, and Yoji Kuroda, "Vision Based Far-Range Perception and Traversability Analysis using Predictive Probability of Terrain Classification," in *Proceedings of International Symposium on Robotics*, Munich, 2010.

[47] Soumya Ghosh and Jane Mulligan, "A Segmentation Guided Label Propagation Scheme for Autonomous Navigation," in *Proceedings of IEEE International Conference on Robotics and Automation*, Anchorage, 2010.

[48] Michael Happold, Mark Ollis, and Nik Johnson, "Enhancing Supervised Terrain Classification with Predictive Unsupervised Learning," in *Proceedings of Robotics Science and Systems Conference*, Philadelphia, 2006.

[49] Dongshin Kim, Jie Sun, Sang Min Oh, James M. Rehg, and Aaron F. Bobick, "Traversability Classification using Unsupervised On-line Visual Learning for Outdoor Robot Navigation," in *Proceedings of IEEE International Conference on Robotics and Automation*, Orlando, 2006.

[50] Anelia Angelova, Larry Matthies, Daniel Helmick, and Pietro Perona, "Dimensionality Reduction Using Automatic Supervision for Vision-Based Terrain Learning," in *Proceedings of Robotics Science and Systems*, Atlanta, 2007.

[51] K. Konolige et al., "Mapping, Navigation, and Learning for Off-Road Traversal," *Journal of Field Robotics*, vol. 26, no. 1, pp. 88-113, 2009.

[52] Visual Studio. Last accessed in May, 2013 [Online]. http://www.microsoft.com/visualstudio/eng/visual-studio-update

[53] Welcome - OpenCV Wiki. Last accessed in December, 2012 [Online]. http://opencv.willowgarage.com/wiki/

[54] AForge.NET : Computer Vision, Artificial Intelligence, Robotics. Last accessed in November, 2012 [Online]. http://aforgenet.com/

[55] Point Grey Bumblebee®2. Last accessed in , 2010 [Online]. http://ww2.ptgrey.com/stereo-vision/bumblebee-2

[56] Google Maps. Last accessed in May, 2013 [Online]. https://maps.google.com/

[57] "Technical Application Note (TAN2008005): Stereo Vision Introduction and Applications," Point Grey Research, 2010.

[58] Point Grey - Stereo Vision Products - Triclops SDK. Last accessed in May, 2013 [Online]. http://www.ptgrey.com/products/triclopsSDK/index.asp

[59] File:HSV_cylinder.png - Wikimedia Commons. Last accessed in May, 2013 [Online]. http://commons.wikimedia.org/wiki/File:HSV_cylinder.png

[60] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of Royal Statistical Society*, vol. 39, no. 1, pp. 1-38, 1977.

[61] Hierarchical clustering simple diagram. PNG Image. Last accessed in May, 2013 [Online]. http://en.wikipedia.org/wiki/File:Hierarchical_clustering_simple_diagram.svg

[62] M. A. Vogel and A. K. C. Wong, "PFD Clustering Method," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 1, no. 3, pp. 237-245, March 1979.

[63] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*, 2nd ed.: John Wiley & Sons, 2001.

[64] C. D. Cantrell, *Modern Mathematical Methods for Physicists and Engineers*.: Cambridge University Press, 2000.

[65] Histograms. Last accessed in May, 2013 [Online]. http://docs.opencv.org/modules/imgproc/doc/histograms.html?

[66] Parallel Processing and Computing Platform | CUDA | NVIDIA. Last accessed in may, 2013 [Online]. http://www.nvidia.com/object/cuda_home_new.html

[67] OpenCL - The open standard for parallel programming of heterogeneous systems. Last accessed in May, 2013 [Online]. http://www.khronos.org/opencl/

[68] Developing games (preliminary). Last accessed in May, 2013 [Online]. http://msdn.microsoft.com/library/windows/apps/hh452744.aspx

[69] V. Volkov, "Better performance at lower occupancy," in *Proceedings of GPU Technology Conference*, 2010.

[70] Mark Harris. Optimizing Parallel Reduction in CUDA. Presentation File. Last accessed in , 2008 [Online]. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

[71] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, pp. 1526-1538, 1987.

[72] M. Harris. Parallel Prefix Sum (Scan) with CUDA. Document. Last accessed in February, 2007 [Online]. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

[73] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel Graph Component Labelling with GPUs and CUDA," *Parallel Computing*, vol. 36, no. 12, pp. 655-678, December 2010.

[74] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider, "Connected component labeling on a 2D grid using CUDA," *Journal of Parallel and Distributed Computing*, vol. 71, no. 4, pp. 615-620, October 2011.

[75] A. D. Pangborn, "Scalable Data Clustering using GPUs," Rochester Institute of Technology, Rochester, M.S. Thesis 2010.

[76] R. A. Dilan, "Unstructured Road Recognition and Following for Mobile Robots via Image Processing using ANNs," Middle East Technical University, Ankara, M.S. Thesis 2010.